UVM Framework Users Guide

Revision 3.6h

UVMF Users Guide Table of Contents

1 Introduction to the UVM Framework (UVMF)

1.1 Motivation for using UVMF

1.1.1 UVM Jumpstart

The steep learning curve of UVM often prevents product teams from realizing the productivity and quality benefits of using advanced verification methodology. The UVM Framework (UVMF) provides a jump-start for learning UVM and building UVM verification environments. It defines an architecture and reuse methodology upon UVM, enabling teams new to UVM to be productive from the beginning while coming up the UVM learning curve. The python scripts provided automate the creation of the files, infrastructure and interconnect for interface packages, environment packages and project benches. Once generated, developers can promptly focus on adding functionality specific to the design and interfaces used.

1.1.2 Reuse Methodology

The UVM Framework is a model reuse methodology that verification teams can leverage. It supports component level verification reuse across projects and environment reuse from block through chip to system level simulation. The UVM Framework is an established verification reuse methodology that is in use at many companies in multiple industries across North America and Europe.

1.1.3 Single Architecture for Simulation and Emulation

The UVM Framework provides an architecture that supports pure simulation and accelerated simulation using emulation. This enables the creation of a single unified environment that supports block, chip and system level tests, and with the choice of running on a pure simulation platform (e.g. Questa) or a hardware-assisted acceleration platform using emulation (e.g. Veloce).

1.1.4 Coverage Within UVMF

UVMF provides a mechanism for rapid creation of reusable simulation infrastructure. Coverage collection components can be defined and connected in the environment using the UVMF code generators. The list below identifies components where functional coverage can be collected and how to add coverage to these components:

1. Coverage component: Create this component using the defineAnalysisComponent API and connect to other components in the environment generator. It will likely have all analysis exports and no analysis ports.
2. Predictor: Manually add required cover groups to the predictor that was generated using defineAnalysisComponent.

3 Scoreboard: Extend UVMF scoreboards on a per-environment basis to define cover groups and sample coverage based on DUT output transactions.

It is important to collect coverage on data that has been validated. In order to avoid agent coverage being confused with coverage of scoreboard validated data the default value of the has_coverage bit in the uvmf_parameterzed_agent is 0. This will prevent the coverage component within the agent from being constructed. Users will have to manually change this bit to enable agents to record transaction coverage at the interface agent.

The UVMF scoreboards contain features that help avoid the falsely optimistic level of coverage:

1 end_of_test_activity_check: This flag is set by default. It generates a uvm error if no transactions were received. This will help avoid mistakes that result in the scoreboard not being attached to prediction or prediction not sending expected transactions. This flag can be set for specific scoreboards in the build_phase of the environment.
2 end_of_test_empty_check: This flag is set by default. It generates a uvm error if transactions remain in the scoreboard in the check_phase. This will help avoid mistakes that result in no transactions being received for comparison against expected transactions. This flag can be set for specific scoreboards in the build_phase of the environment.
3 wait_for_scoreboard_empty: This flag is clear by default. It postpones the termination of run_phase until there are no transactions in the scoreboard. The UVM based timeout, UVM_DEFAULT_TIMEOUT, is used to prevent simulations from hanging.
4 The uvm messaging ID field of UVMF scoreboards contain the scoreboard hierarchy. A cursory view of the message summary at the end of the transcript will identify the absence of a scoreboard.

1.2 Major Divisions of Functionality Within UVMF

1.2.1 UVMF Base Package

The UVMF base package, `uvmf_base_pkg`, is a library of base classes that implement core functionality of components found in all simulation benches. This includes base classes for transactions, sequences, drivers, monitors, predictors, scoreboards, environments and tests. All classes in the UVMF base package are derived from UVM classes. User extensions then provide variables, tasks and functions specific to the design under test.

The UVMF base package and the package structure used within UVMF define a UVM reuse methodology. This methodology supports horizontal reuse, i.e. reuse of components across projects, as well as vertical reuse, i.e. environment reuse from block to chip to system.

1.2.2 Interface Packages

UVMF interface packages and their associated BFMs provide all of the functionality required to monitor and optionally drive a design interface. Interface packages and BFMs are reusable across projects. An interface package is composed of three pieces: a signal bundle interface, BFM interfaces and the package declaration. The signal bundle contains all signals used in the protocol. The BFMs implement the protocol signaling to drive and monitor transactions at the pin level. The package declaration includes all class definitions and type definitions used by the interface agent.

1.2.3 Environment Packages

The environment package is a key aspect that enables vertical reuse of environments within the UVMF. The environment package contains the environment class definition, its configuration class definition and any environment level sequences that could be used in higher level simulations. Block level environments contain agents, predictors, scoreboards, coverage collectors and other components. All other levels of environment include other environments. Environments are structured hierarchically similar to the way RTL is composed hierarchically.

1.2.4 Verification IP

The verification_ip folder contains all packages that are reused across projects and from block to top. This folder contains environment packages, interface packages, utility packages, etc.

1.2.5 Project Benches

The simulation bench is composed of top level elements that are not generally intended to be reusable horizontally nor vertically. It defines test level parameters, the top level modules, top level sequence and top level UVM test. It also includes derived sequences and tests used to implement additional test scenarios.

1.2.6 Example Groups

UVMF examples are divided into groups. The groups include base_examples, vip_examples and vista_examples. The base examples are the core examples and run in simulation and emulation. The vip examples contain a Questa VIP and emulatable VIP example for AXI4. A Questa VIP license and software installation is required in addition to a Questa license to run the QVIP example. A Veloce software license and software installation is required in addition to a Questa license to run the VIP example. The Vista example requires a Vista license and software installation in addition to a Questa license to run.

Each example group contains a verification_ip and project_benches folder. The verification_ip folder contains all reusable packages used and shared among benches in the project_benches folder. The benches can also use packages found in the verification_ip folder in the base_examples group.

1.3 UVMF Base Class Package Overview

1.3.1 Overview

The `uvmf_base_pkg`, located under the `verification_ip` directory, provides a library of classes that implements the methodology used by the UVMF. In order to support emulation UVMF is divided into two packages: uvmf_base_pkg and uvmf_base_pkg_hdl. The latter only contains the synthesizable typedefs and parameters required by the emulated portion of a test bench. The former includes all class definitions and other non-synthesizable typedefs. The uvmf_base_pkg_hdl package is imported by uvmf_base_pkg. Each class within uvmf_base_pkg is described below.

1.3.2 Stimulus Classes

1.3.2.1 uvmf_transaction_base.svh

This is the base class for all sequence items, i.e. transactions, within UVMF. It provides a unique transaction ID variable and associated functions useful for debug. It also provides variables used for transaction viewing and a unique key for storing the transaction in associative arrays.

1.3.2.2 uvmf_sequence_base.svh

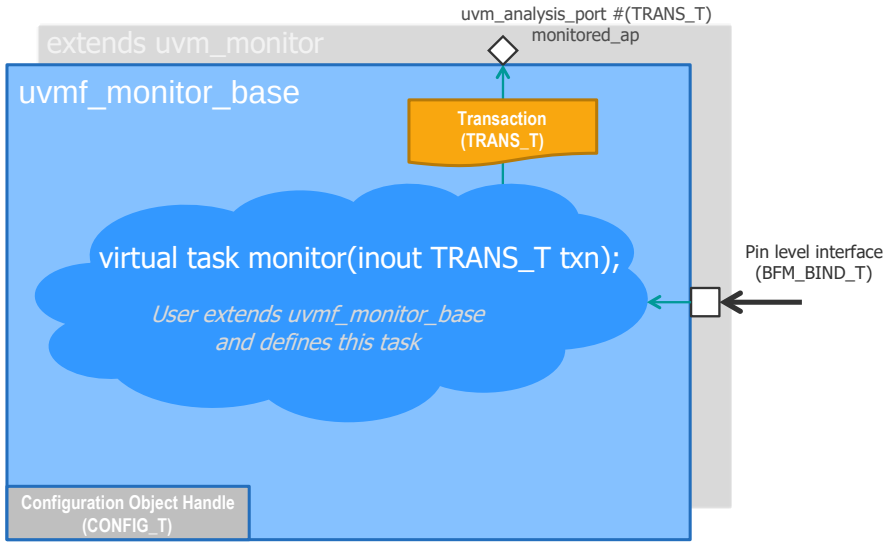
This is the base class for all sequences within UVMF. It extends uvm_sequence but provides no additional functionality. It provides a location where functionality common to all UVMF sequences can be added.

1.3.3 Agent Classes

1.3.3.1 uvmf_monitor_base.svh

This is the base class for all UVMF monitors. Only monitors extended from uvmf_monitor_base should be used as specialization of the MONITOR_T parameter of the uvmf_parameterized_agent base class. When extending the uvmf_monitor_base, only the monitor task must be defined. The monitor task either observes and captures bus activity directly or calls a task in the monitor BFM which captures bus activity. It is recommended that signal level bus monitoring be done in the monitor BFM for optimal run-time performance, especially with emulation.

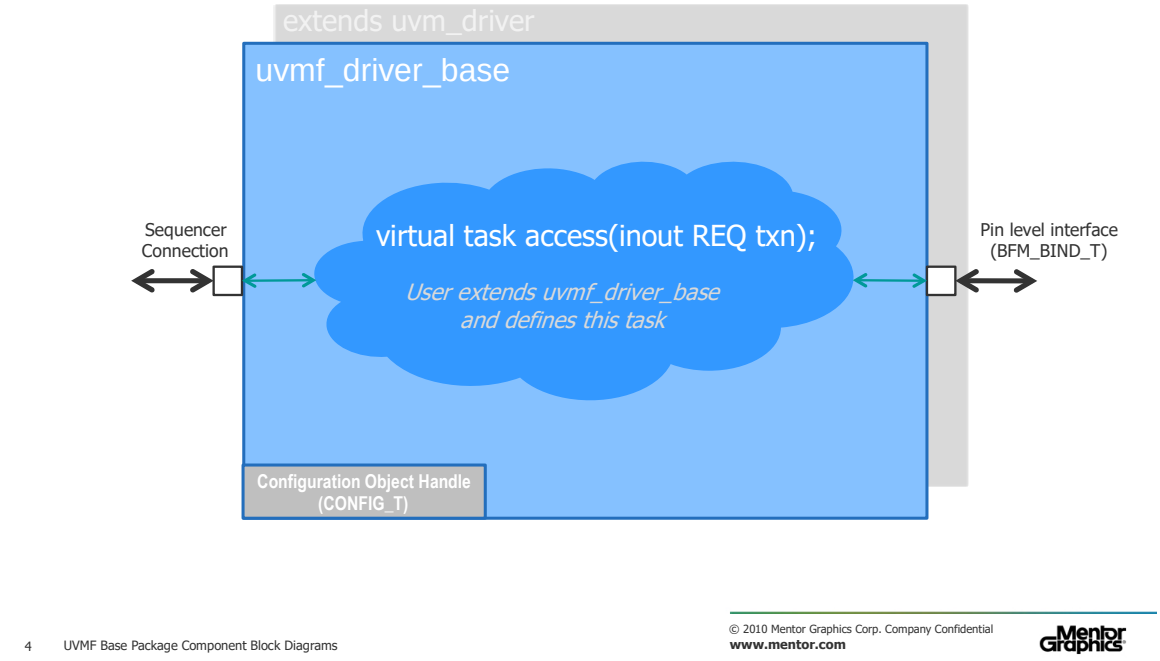
UVMF Monitor Base



3 UVMF Base Package Component Block Diagrams

1.3.3.2 uvmf_driver_base.svh

This is the base class for all UVMF drivers. Only drivers extended from uvmf_driver_base should be used as specialization of the DRIVER_T parameter of the uvmf_parameterized_agent base class. When extending uvmf_driver_base, only the access task must be defined. The access task either drives bus activity directly or calls a task in the driver BFM which drives activity. It is recommended that signal level bus driving be done in the driver BFM for optimal run-time performance, especially with emulation.



1.3.3.3 uvmf_parameterized_agent_configuration_base.svh

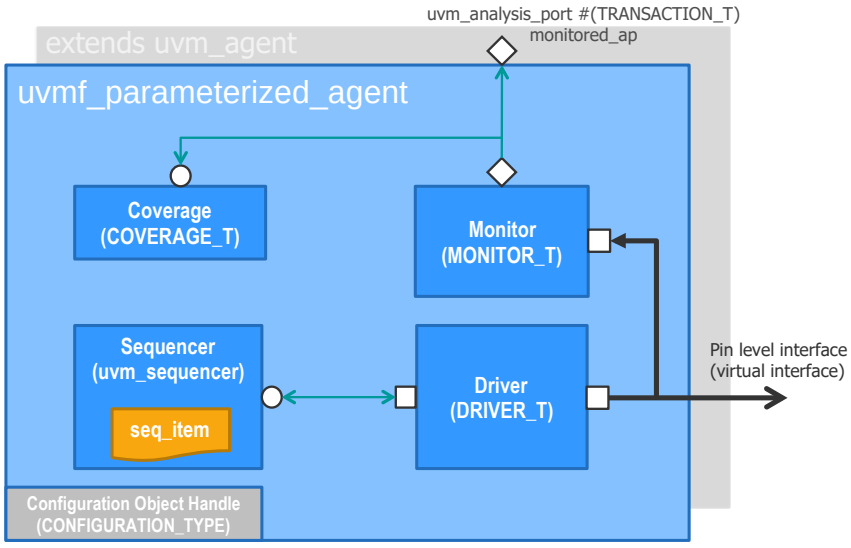
This is the base class for all interface agent configurations. Only configurations extended from uvmf_parameterized_agent_configuration_base should be used as specialization of the CONFIG_T parameter of the uvmf_parameterized_agent base class. Variables common to all agents and specific to the uvmf_parameterized_agent are in the uvmf_parameterized_agent_configuration_base. Add protocol specific configuration variables to an extension of uvmf_parameterized_agent_configuration_class.

1.3.3.4 uvmf_parameterized_agent.svh

This class implements an interface agent. This agent can be used with any protocol. The protocol specific features reside in the configuration, driver, monitor, coverage and transaction types used as parameters to this class. If the agent is active then it automatically places its sequencer within the uvm_config_db for retrieval by the top level sequence.

Prior to constructing a monitor, the parameterized agent checks the uvm_config_db for a monitor of the required type. If one is returned then it is used instead of constructing a local monitor. This is to support construction of a shared monitor in an upper level environment. The shared monitor is created in an upper level environment where lower level environments monitor common interfaces.

UVMF Parameterized Agent



32 Your Initials, Presentation Title, Month Year

1.3.4 Predictor Classes

The use of UVMF predictor classes, uvmf_predictor_base and uvmf_sorting_predictor_base has been replaced by application specific predictors and other analysis components that can be generated using the UVMF code generators.

The UVMF environment generator provides a way for users to specify analysis components with any combination of analysis_exports and analysis_ports. This allows the user to automatically generate any analysis component required. One type of analysis component that can be generated are predictors. Predictor specification for the UVMF code generator includes the number and type of analysis_exports required as well as the number and type of analysis_ports required. Another type of analysis component that can be generated are coverage components. Coverage component specification for the UVMF code generator includes the number and type of analysis_exports required. Coverage components typically do not have analysis_ports so the list of analysis_port type and names would be empty.

1.3.5 Scoreboard Classes

The scoreboards provided within UVMF perform comparison between predicted and actual DUT output transactions. UVMF scoreboards perform no prediction operations. Within UVMF all prediction is performed using predictors. This allows reuse of scorebaords. The use of UVMF in-order scoreboards require the transaction class being compared contain a compare function. The use of UVMF out-of-order scoreboards require the transaction class being compared contain a compare function and get_key function.

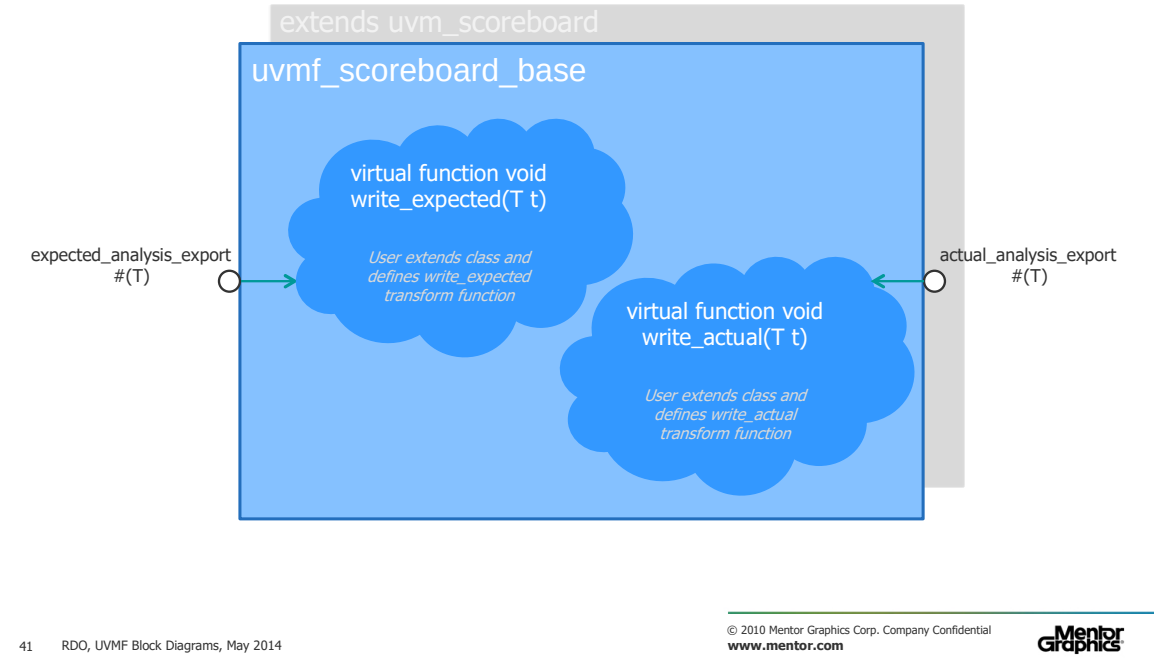
All UVMF scoreboards provide an end of test empty check. This is enabled using the end_of_test_empty_check flag in the uvmf_scoreboard_base class. If enabled, the scoreboard will check for remaining transactions in the scoreboard in the check_phase. If transactions exist and the flag is set a UVM Error is generated and a programmable number of transactions are displayed in the transcript.

All UVMF scoreboards provide an end of test activity check. This is enabled using the end_of_test_activity_check flag in the uvmf_scoreboard_base class. If enabled, in the check_phase a UVM error is generated if no transactions have been received through the expected analysys_export.

All UVMF scoreboards provide a mechanism for delaying the end of the run_phase until all transactions have been drained from the scoreboard. This is enabled using the wait_for_scoreboard_empty flag. If enabled, at the end of the run_phase the scoreboard is checked for remaining transactions. If transactions remain then the scoreboard raises an objection until all transactions have been drained from the scoreboard.

1.3.5.1 uvmf_scoreboard_base.svh

This is the base class for all scoreboards within the UVM Framework. It provides the two analysis exports for receiving transactions, expected_export and actual_export. It also provides basic end of test use checks and reporting.

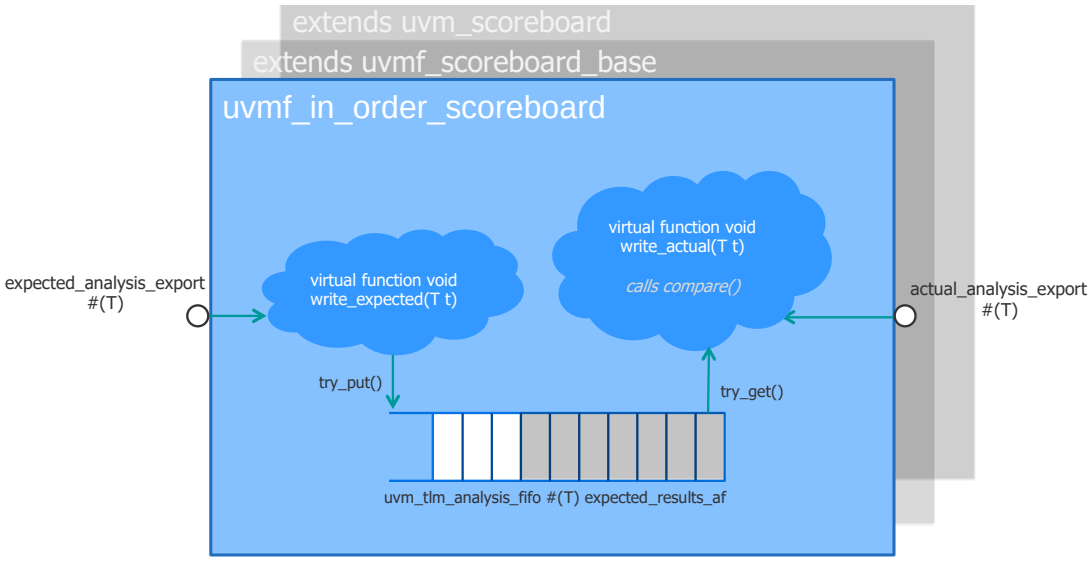


1.3.5.2 uvmf_in_order_scoreboard.svh

This scoreboard is used in circumstances where the data order through the DUT is preserved.

The in order scoreboard extends the scoreboard base. It adds an analysis FIFO for storing expected results. Transactions received through the expected_export are placed into the analysis FIFO. Transactions observed on the DUT output are sent to the actual export for comparison. The arrival of a transaction on the actual export causes the next transaction to be removed from the analysis FIFO and compared to the actual transaction. An error is generated if the FIFO is empty.

UVMF In-Order Scoreboard

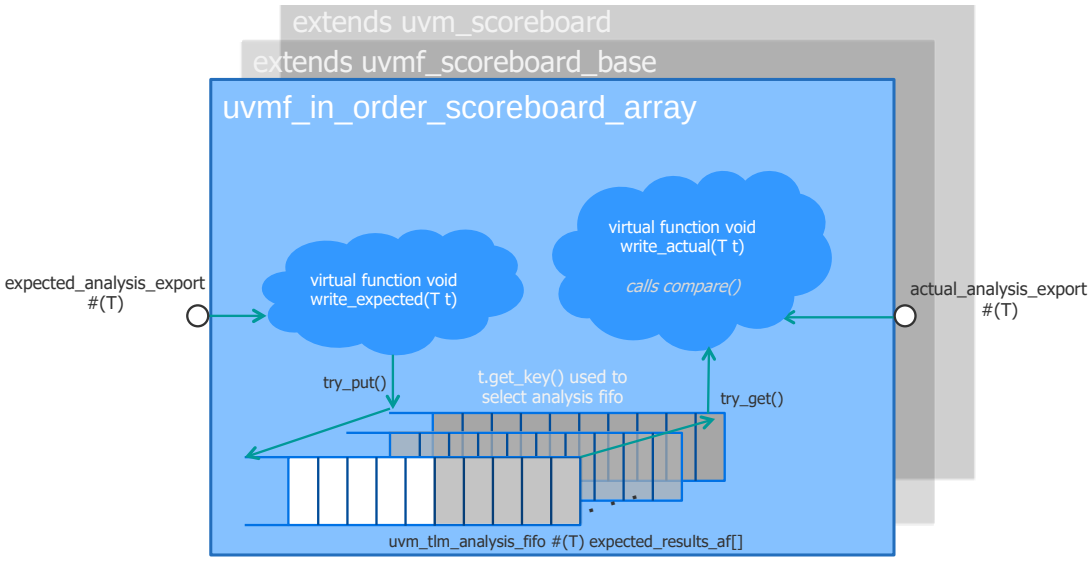


1.3.5.3 uvmf_in_order_scoreboard_array.svh

This scoreboard is used in circumstances where data through a physical channel is divided into multiple logical channels and data order within a logical channel is in order.

The in order scoreboard array implements an analysis FIFO for each logical channel. The behavior for each channel is identical to the in order scoreboard. The logical channel for each incoming transaction is identified using the get_key function of the transaction.

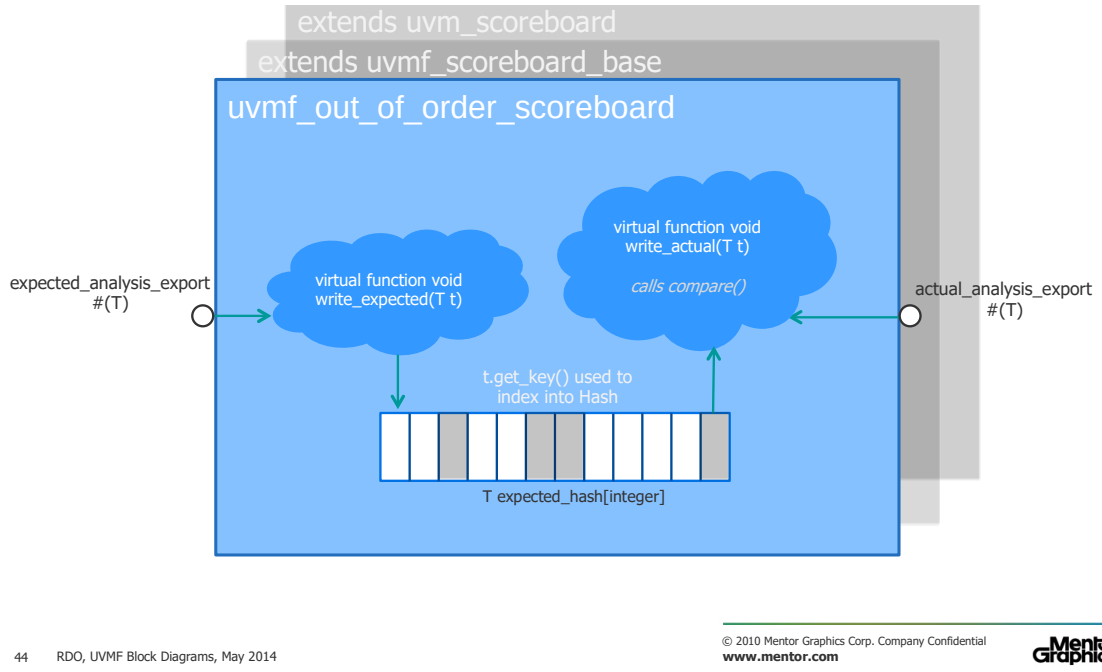
UVMF In-Order Scoreboard Array



1.3.5.4 uvmf_out_of_order_scoreboard.svh

The out of order scoreboard is used in circumstances where data order through the DUT is not guaranteed or not predictable.

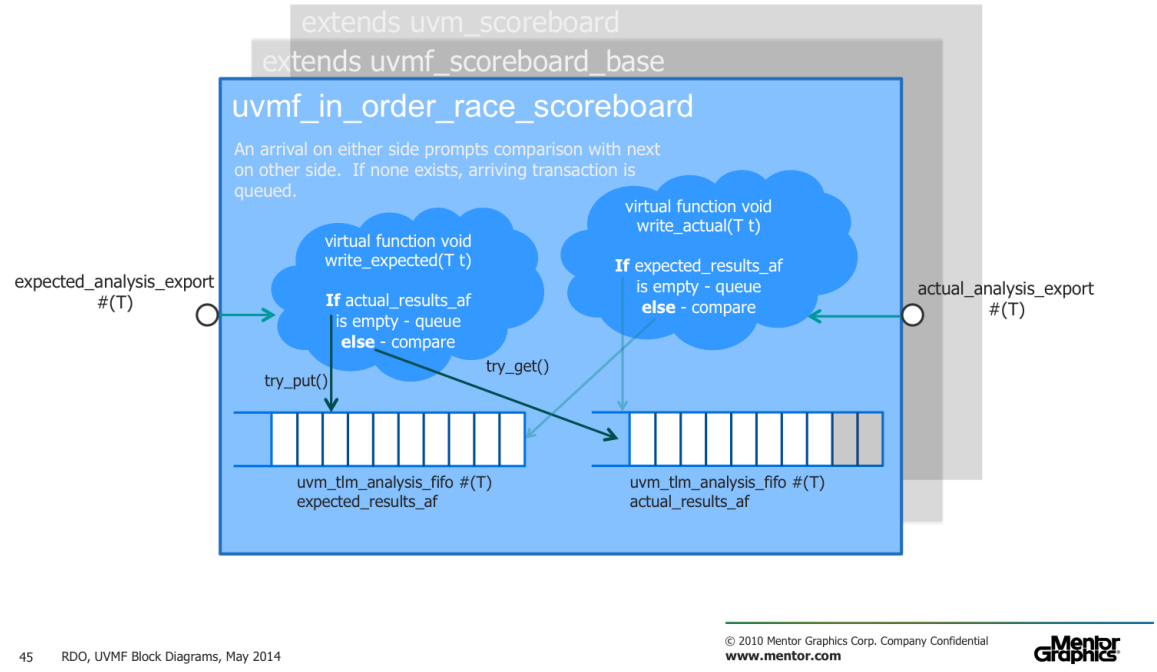
The out of order scoreboard extends the scoreboard base. It adds a SystemVerilog associative array for storing expected results. Transactions received through the expected_export are placed into the associative array using the value returned from the get_key function as the key of the entry. Transactions observed on the DUT output are sent to the actual export for comparison. The arrival of a transaction on the actual export causes a transaction to be removed from the associative array and compared to the actual transaction. The get_key function of the actual transaction is used to identify a matching transaction in the associative array. An error is generated if the associative array does not have an entry that matches the key returned from the actual transactions get_key function.



1.3.5.5 uvmf_in_order_race_scoreboard.svh

The in order race scoreboard is used in circumstances where data order through the DUT is preserved and the DUT can send output transactions before input transactions are received.

The in order race scoreboard extends the scoreboard base. It adds an analysis FIFO for the expected_export and actual_export. When a transaction arrives on either port the other port is checked for an entry to compare against. If an entry exists in the FIFO for the other port then the entry is pulled from the FIFO for comparison. If an entry does not exist in the FIFO for the other port then the entry is queued for later comparison.



1.3.6 Environment Classes

1.3.6.1 uvmf_environment_configuration_base.svh

The uvmf_environment_configuration_base is the base environment configuration class for all UVMF environments. It provides flags used for register model integration. It also provides debug features for the initialize call and its handling of interface name and activity arrays.

1.3.6.2 uvmf_environment_base.svh

The uvmf_environment_base is the base class for all UVMF based environments. It provides a handle to the environments configuration class.

1.3.6.3 Parameterized Environments

The parameterized environments contained in the uvmf_base_pkg are no longer recommended. It is recommended that users generate design specific environments using the UVMF environment generator. The parameterized environments include uvmf_parameterized_*_environment classes.

1.3.7 Test Classes

1.3.7.1 uvmf_test_base.svh

This is the base class for the base test for all simulation benches. The uvmf_test_base instantiates the top level configuration, top level environment and top level sequence. The test class directly extended from uvmf_test_base is named test_top and must define the parameters for the top level configuration, environment and sequence and calls the initialize function of the configuration class. Test top is then extended to create additional test cases by specifying factory overrides.

1.4 UVMF Interface Overview

Introduction

The interface used to implement a protocol is divided into three pieces: driver BFM, monitor BFM and signal bundle. Each of these pieces is a SystemVerilog interface. The driver BFM interface provides the tasks and functions needed to drive signals according to the protocol specification. The monitor BFM interface provides the tasks and functions needed to observe the interface protocol and capture relevant information from each bus transfer. The signal bundle interface contains all signals used by the protocol. It is also where assertions for protocol checking should reside.

The signals in the signal bundle interface are all of net type, either wire, tri, tri0 or tri1. The signals in this interface cannot be defined as types with storage, bit, reg or logic in order to enable binding the signal bundle into the design where signals are driven by RTL.

The signals are separated into an interface to support block to top reuse. The signal bundle interface is either hierarchically connected or bound into the desired level of hierarchy when the bus to be observed is driven by RTL instead of a driver_BFM. The signal bundle interface is connected to a monitor BFM and optionally a driver BFM through the BFMs port list. When the signal bundle is bound into the design the signal bundle is passed to the BFMs by hierarchical reference.

The diagrams below describe the function of and interaction between the signal bundle interface, the BFM interface and the UVM component. It is important to note that though the examples use a SystemVerilog virtual interface handle as the connection between the BFM and UVM component, the BFM_BIND_T parameter of the UVM component allows for alternative models of communication with the BFM, such as the DPI-C based function call model which requires instead the use of SystemVerilog hierarchical scopes stored as chandles to establish the connection between BFM and UVM component

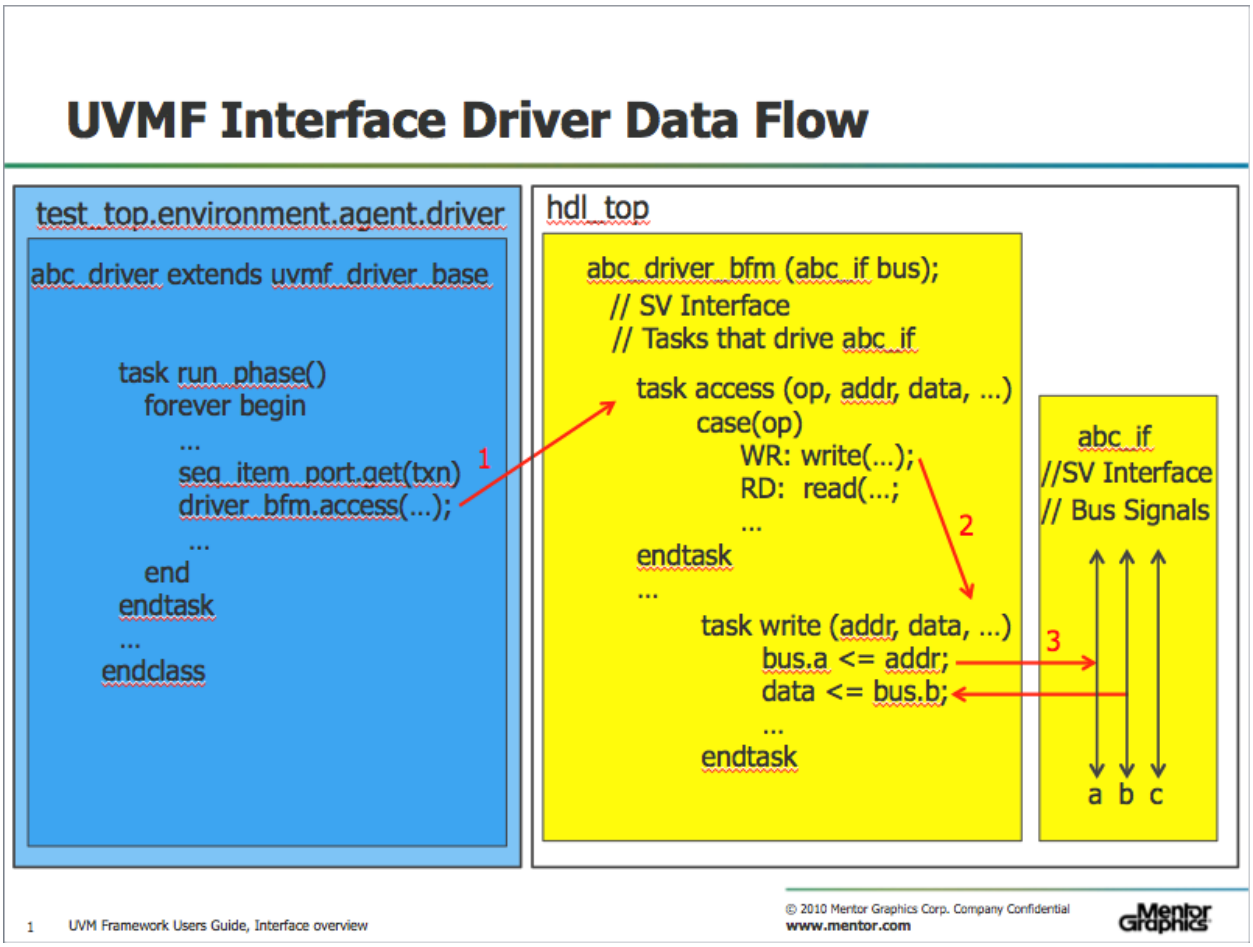
1.4.1 Interface Driver BFM Flow

The following diagram shows the function of and interaction between the UVM driver, driver BFM and signal bundle interface.

The abc_driver is a UVM component derived from uvmf_driver_base. Its role is to request sequence items from the sequencer and send pertinent transaction information to the driver BFM. The sequence item handle can be sent directly from the abc_driver to the abc_driver_bfm. Alternatively, in support of emulation, the arguments to the access task must be synthesizable. In this case individual data attributes from the sequence item can be passed as arguments to the access task or the data attributes can be encapsulated into a packed struct that is passed to the abc_driver_bfm through the access task.

The abc_driver_bfm is a SystemVerilog interface that drives the bus signals according to the protocol specification. The content of this interface needs to be synthesizable in order to support emulation. In emulation, the BFMs as along with the DUT are synthesized and loaded into the emulator. Access to the signal bundle interface is provided to the driver BFM through its port list. The driver BFM drives the signals in the signal bundle interface.

The signal bundle interface contains all signals used in the protocol. It is used to connect the DUT to the monitor BFM and optionally the driver BFM.



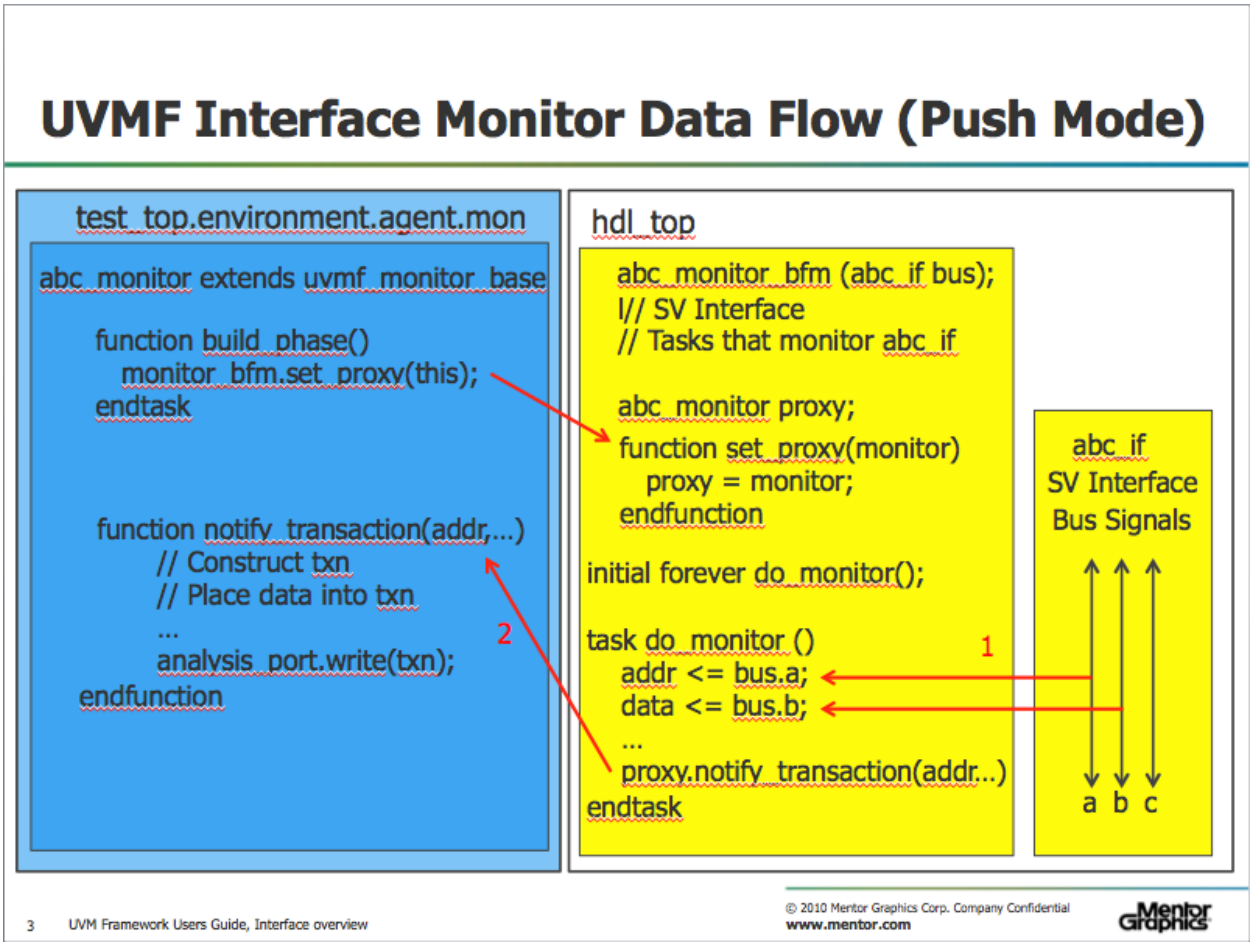
1.4.2 Interface Monitor BFM Flow

1.4.2.1 Push Mode Interface Monitor BFM Flow

The following diagram shows the function of and interaction between the UVM monitor, monitor BFM and signal bundle interface for a monitor using the s-called push mode for data transfer between the monitor BFM and UVM monitor. In the push mode the UVM BFM passes, or 'pushes', data unidirectionally to the UVM monitor. That is, communication only happens from the monitor BFM to the UVM monitor and this is done in zero simulation time since a function in the UVM monitor is used to receivethe pushed data. Thecommunication overhead between the abc_monitor and abc_monitor_bfm is minimized which can significantly increase in particular emulation performance.

The abc_monitor is a UVM component derived from uvmf_monitor_base. Its role is to broadcast sequence items received from the monitor BFM to other UVM components in the environment. Its handle in the monitor BFM is set using the set_proxy function in the monitor BFM during the UVM build_phase. This handle allows the monitor BFM to call functions in the UVM monitor to push data to the monitor.

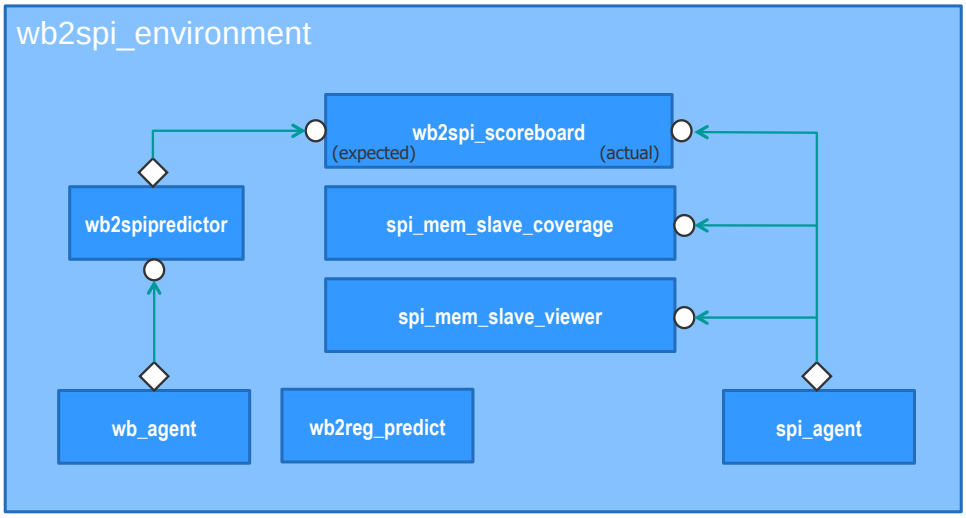
The abc_monitor_bfm is a SystemVerilog interface that observes the bus signals according to the protocol specification. The content of this interface needs to be synthesizable in order to support emulation. In emulation, the BFMs as well as the DUT are synthesized and loaded into the emulator. Access to the signal bundle interface is provided to the monitor BFM through its port list. The monitor BFM observes the signals in the signal bundle interface. Once the abc_monitor_bfm has observed a complete transfer on the bus it pushes the data observed to the abc_monitor using the notify_transaction function. The notify_transaction function in the abc_monitor copies the data to a transaction object and broadcasts the transaction to other components in the environment.



1.5 UVMF Environment Overview

The diagram below shows a block level environment. It is from the WB2SPI example. Block level environments include agents, predictors, scoreboards, coverage collectors and other components that are connected based on design data flow.

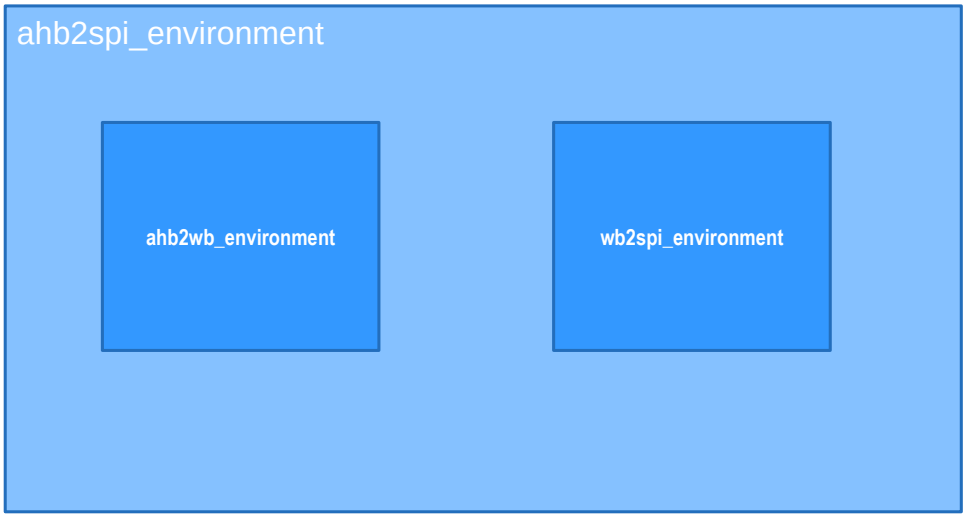
WB2SPI Example: Environment



WB2SPI Example Block Diagrams

The diagram below shows a chip level environment. It is from the AHB2SPI example. Chip level environments include other environments. This is true for all environments other than block level environments. Environments are structured in a hierarchical manner similar to how RTL blocks are composed hierarchically. This allows for environment reuse as RTL components are reused.

AHB2SPI Example: Environment



WB2SPI Example Block Diagrams

1.6 UVMF Simulation Bench Overview

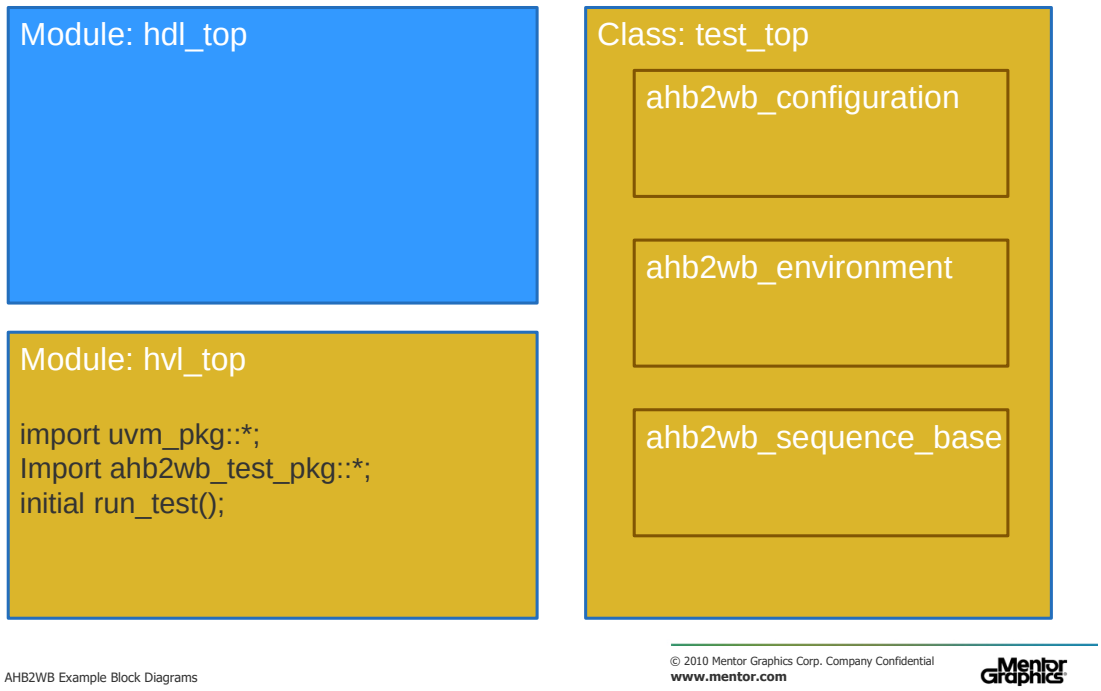
The structure of the simulation bench is shown below. It is from the AHB2WB simulation bench.

The three top level elements in every UVMF simulation bench are hdl_top, hvl_top and test_top.

Synthesizable content that may be run in emulation is placed in hdl_top. This includes the DUT, driver BFMs, monitor BFMs, signal bundle interfaces and any other logic required by the DUT.

The non-synthesizable elements that must be in a module are placed in hvl_top. This includes the test package import and the call to UVM's run_test to start the UVM phases.

The top level configuration, top level environment and top level sequence are placed in test_top. This defines the base test from which all other tests are derived.



2 UVM Framework Examples

2.1 Example Benches

2.1.1 AHB to wishbone Example

The AHB2WB example demonstrates a block level environment. It is located in the base_examples group. This block level environment is reused in the AHB2SPI chip level environment example. This example also demonstrates the use of a parameterized environment. The use of a parameterized environment alleviates the need to write an environment class.

This example demonstrates the following:

1. Block level environment that will be reused at the chip level
2. Use of a parameterized environment
3. Test plan import
4. Merging of test results
5. Generation of a custom coverage report

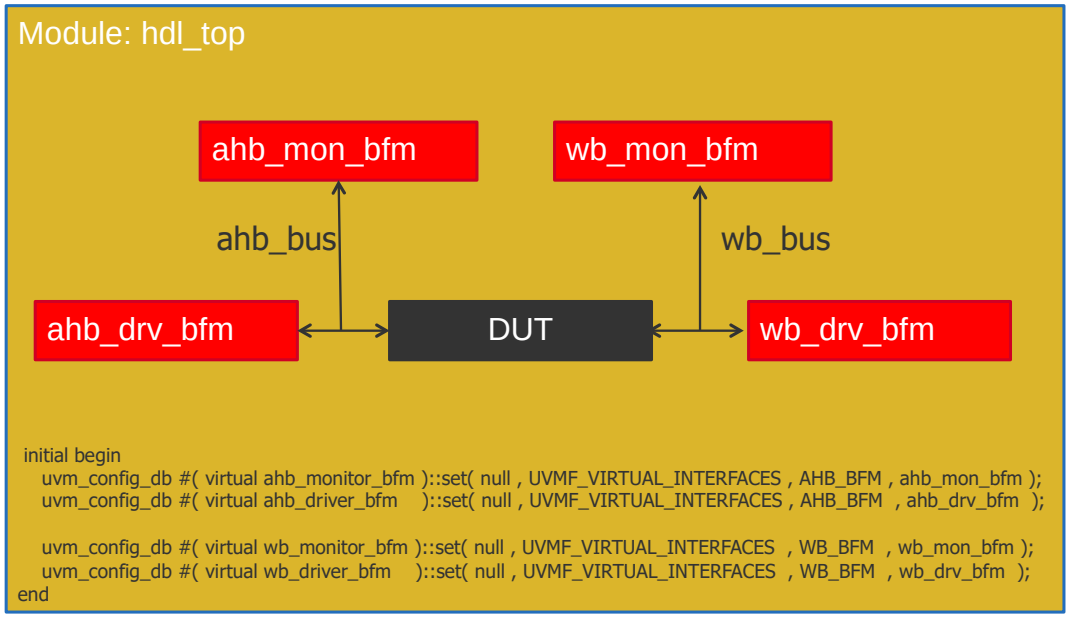
A specification for the ahb2wb DUT can be found in the doc folder of the example.

AHB2WB Example: Test Bench

AHB2WB Example Block Diagrams

As with all UVMF test benches, the AHB2WB test bench is composed of three top levels: hdl_top, hvl_top and test_top. The module named hdl_top contains the DUT, BFMs and signal bundle interfaces that tie them together. All content in hdl_top is synthesizable to support emulation. The module named hvl_top contains all content that must reside within a module but is not synthesizable. This includes importing the test package and calling run_test to start the UVM phases. The class named test_top is the top level UVM test class. It is selected using the +UVM_TESTNAME argument on the command line and constructed by the UVM factory.

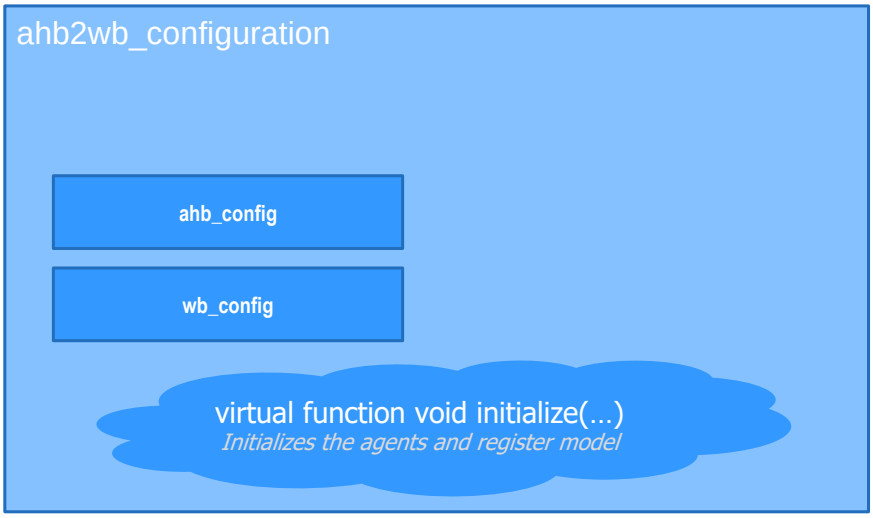
AHB2WB Example: hdl_top



AHB2WB Example Block Diagrams

The hdl_top module contains the DUT and BFMs used to drive and monitor bus activity. The _drv_bfm interfaces provide signal stimulus. The _mon_bfm interfaces observe signal activity and capture transaction information for broadcasting to the environment for prediction, scoreboarding and coverage collection. An interface containing all of the signals for the bus ties the monitor BFM, driver BFM and DUT signal ports together. Protocol signals are separated into an interface to enable block to top reuse of environments and monitor BFMs. All BFMs are placed into the uvm_config_db by hdl_top for retrieval by the appropriate agent configuration. This mechanism is described in the section on resource sharing and initialization within the UVM Framework.

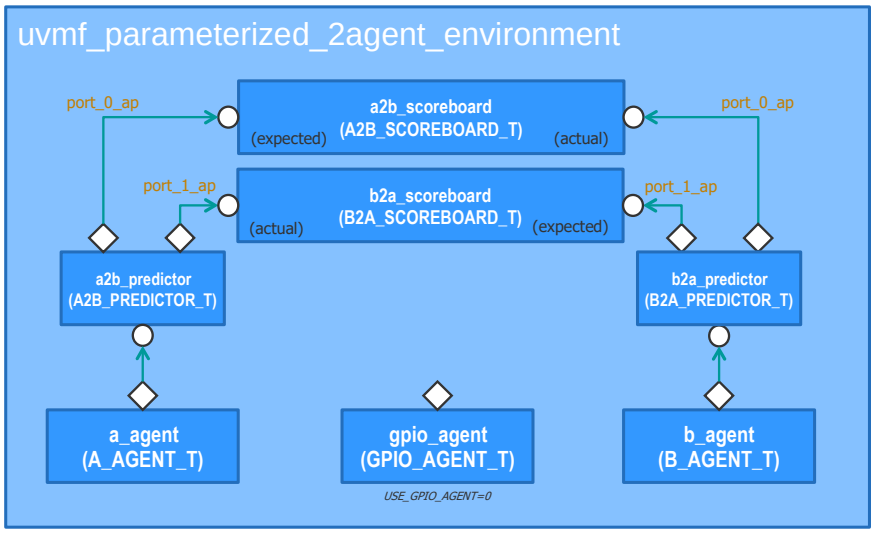
AHB2WB Example: Configuration



AHB2WB Example Block Diagrams

The ahb2wb_configuration class contains a configuration for each agent in the environment. A function named initialize provides the agent configurations with the active/passive state of the agent, the path to its agent in the environment and the string name of the interface to be retrieved from the uvm_config_db. The ahb2wb configuration also contains DUT configuration specific variables that can be randomized as needed. The ahb2wb configuration class is constructed, randomized and initialized before the environment build phase is executed. This ensures that the environment can be built according to the configuration for the simulation.

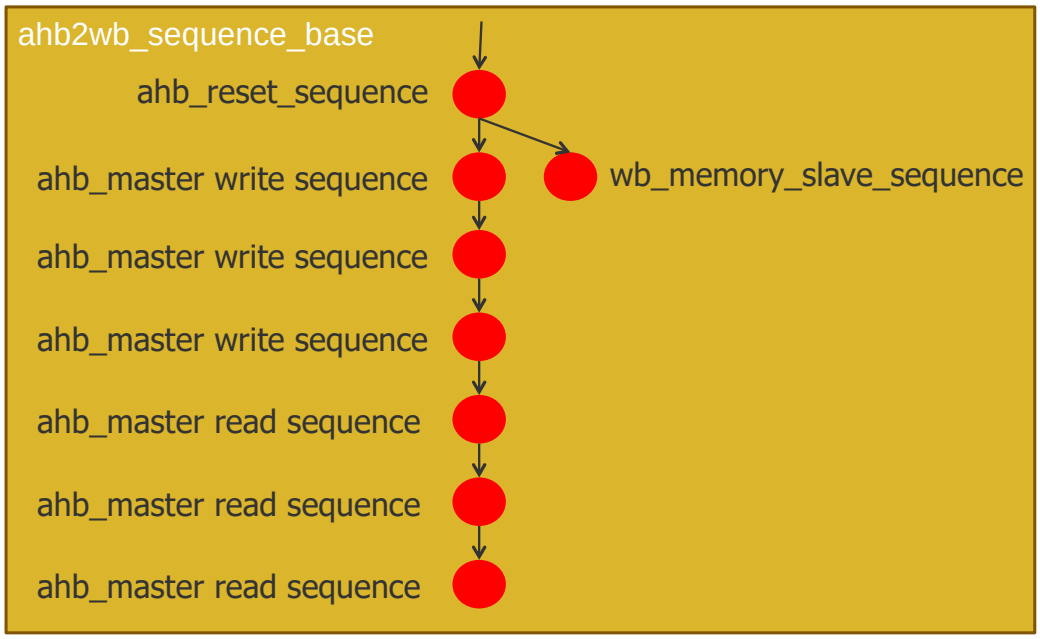
AHB2WB Example: Environment



AHB2WB Example Block Diagrams

The ahb2wb environment utilizes the uvmf_parameterized_2agent_environment. This alleviates the need to write an environment class. The ahb2wb environment is a type specialization of the parameterized environment. Creating a type specialization of the parameterized agents only requires the creation of a typedef. The typedefs used in the ahb2wb example can be found in verification_ip/environment_packages/ahb2wb_env_pkg/src/ahb2wb_environment.svh

AHB2WB Example: Top Level Sequence



AHB2WB Example Block Diagrams

The top level sequence, named ahb2wb_sequence_base, orchestrates and controls all stimulus within the simulation. The stimulus flow is shown in the diagram above. The first sequence to be started is the ahb reset sequence. This causes the ahb_drv_bfm to assert and then release reset. Once the reset sequence has completed the wishbone memory slave sequence is started. This sequence is forked off because it will remain active throughout the simulation. This is because a slave device is always active and ready to respond to activity initiated by the master. Once the slave sequence is forked a series of writes and reads are performed on the ahb2wb DUT.

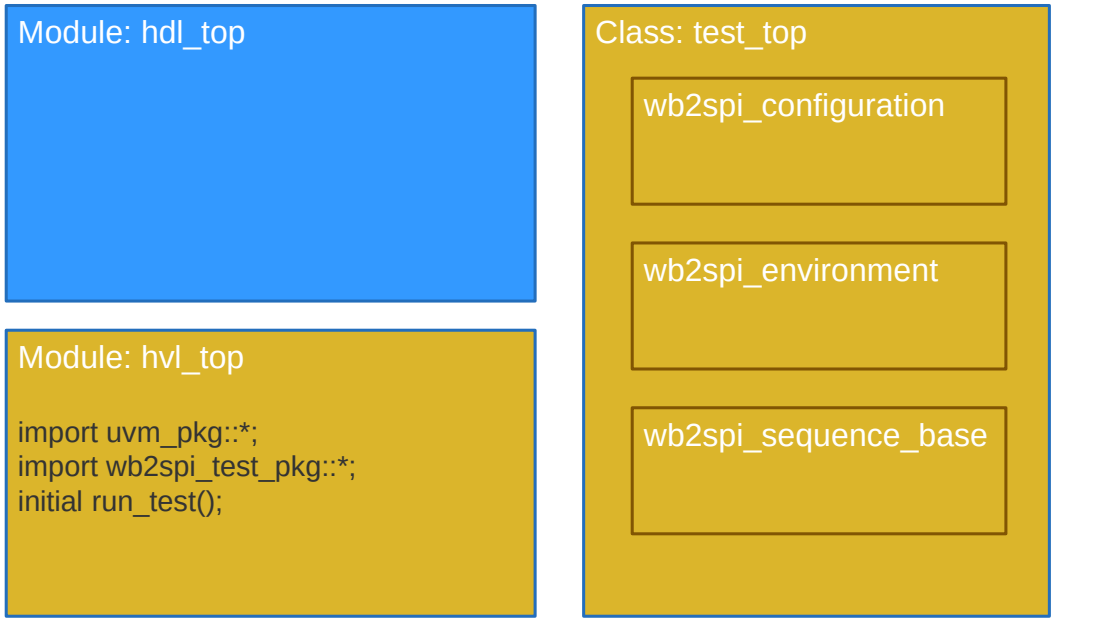
2.1.2 WB to SPI Example

The WB2SPI example demonstrates a block level environment. It is located in the base_examples group. This environment includes a register model based on the UVM register package. This block level environment is reused in the AHB2SPI chip level environment example. A specification for the wb2spi DUT can be found in the doc folder of the example.

This example demonstrates the following:

1. Block level environment that will be reused at the chip level
2. Block level UVM register model that will be reused at the chip level

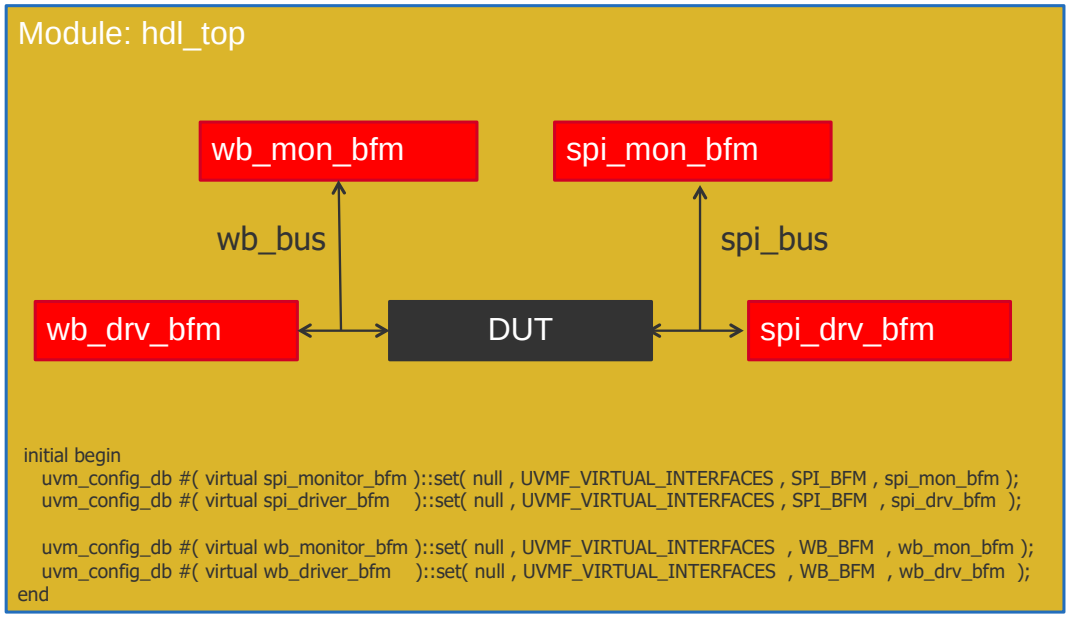
WB2SPI Example: Test Bench



WB2SPI Example Block Diagrams

As with all UVMF test benches, the WB2SPI test bench is composed of three top levels: hdl_top, hvl_top and test_top. The module named hdl_top contains the DUT, BFMs and signal bundle interfaces that tie them together. All content in hdl_top is synthesizable to support emulation. The module named hvl_top contains all content that must reside within a module but is not synthesizable. This includes importing the test package and calling run_test to start the UVM phases. The class named test_top is the top level UVM test class. It is selected using the +UVM_TESTNAME argument on the command line and constructed by the UVM factory.

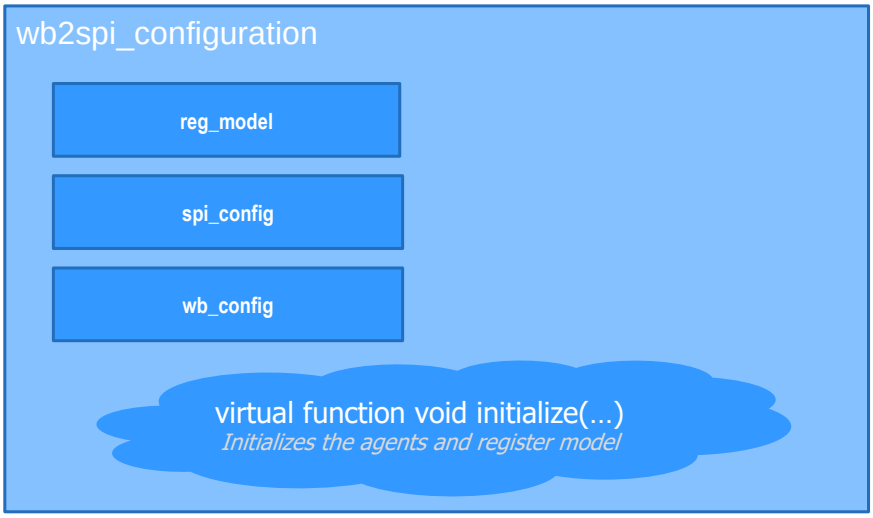
WB2SPI Example: hdl_top



WB2SPI Example Block Diagrams

The hdl_top module contains the DUT and BFMs used to drive and monitor bus activity. The _drv_bfm interfaces provide signal stimulus. The _mon_bfm interfaces observe signal activity and capture signal information for broadcasting to the environment for prediction, scoreboarding and coverage collection. An interface containing all of the signals for the bus ties the monitor BFM, driver BFM and DUT signal ports together. Protocol signals are separated into an interface to enable block to top reuse of environments and monitor BFMs. All BFMs are placed into the uvm_config_db by hdl_top for retrieval by the appropriate agent configuration. This mechanism is described in the section on resource sharing and initialization within UVM Framework.

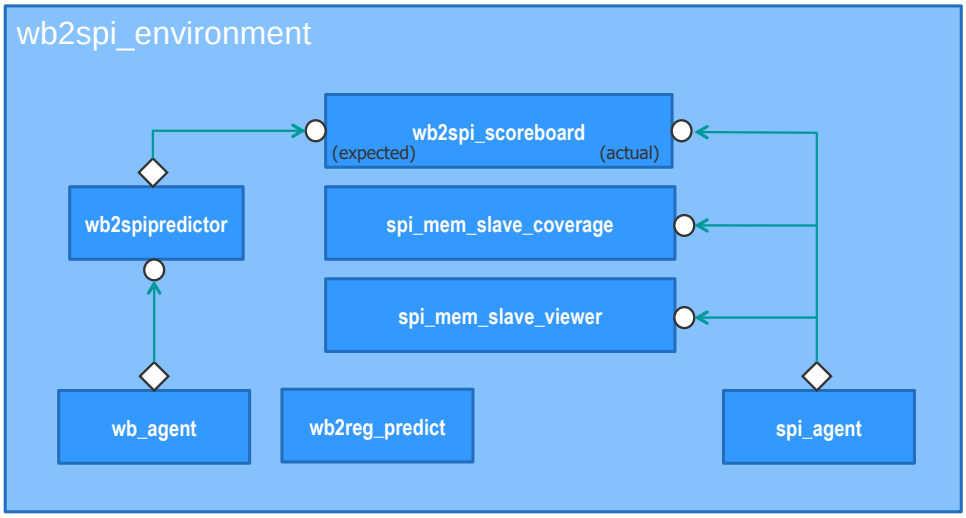
WB2SPI Example: Configuration



WB2SPI Example Block Diagrams

The wb2spi_configuration class contains a configuration for each agent in the environment and the register model. The register model is a UVM register block for the wb2spi DUT. This register block will be a sub block of the ahb2spi register block used in the chip level simulation. A function named initialize provides the agent configurations with the active/passive state of the agent, the path to its agent in the environment and the string name of the interface to be retrieved from the uvm_config_db. The wb2spi configuration also contains DUT configuration specific variables that can be randomized as needed. The wb2spi configuration class is constructed, randomized and initialized before the environment build phase is executed. This ensures that the environment can be built according to the configuration for the simulation.

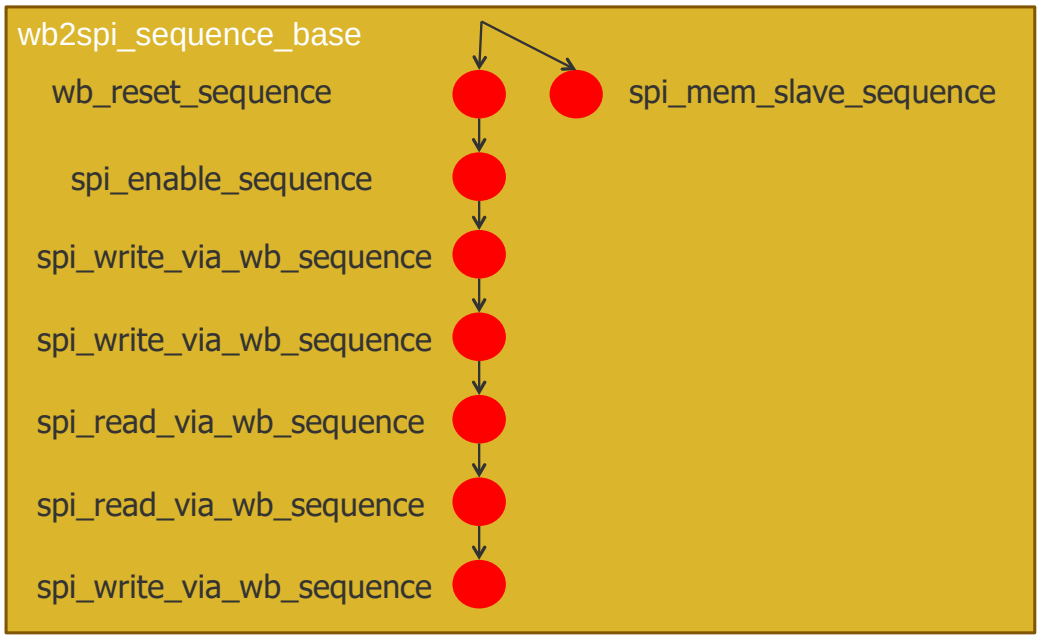
WB2SPI Example: Environment



WB2SPI Example Block Diagrams

The wb2spi environment contains the agents, predictor, scoreboard, coverage and transaction viewing components shown in the above diagram. The wishbone agent interacts with the wb_drv_bfm and wb_mon_bfm. It receives stimulus information from sequences in the top level sequence. Bus operations are observed and broadcasted to the wb2spi predictor. The predictor creates an expected SPI transaction based on DUT configuration and input from the wishbone bus. Output from the wb2spi predictor are sent to the scoreboard to be queued until DUT activity is received for comparison. The SPI agent interacts with the spi_drv_bfm and spi_mon_bfm. It receives stimulus information from sequences in the top level sequence. Bus operations are observed and broadcasted to the wb2spi scoreboard, coverage component and transaction viewing component. The coverage component records functional coverage of SPI operations. The transaction viewing component provides a transaction viewing stream of SPI memory slave transactions. The monitor within the SPI agent provides transaction viewing of the base SPI transfer. This allows for viewing of raw SPI transfers as well as the functional meaning of each bit within the raw SPI transfer.

WB2SPI Example: Top Level Sequence



WB2SPI Example Block Diagrams

The top level sequence, named wb2spi_sequence_base, orchestrates and controls all stimulus within the simulation. The stimulus flow is shown in the diagram above. The first sequence to be started is the SPI memory slave sequence. This sequence is forked off at the beginning because it will remain active throughout the simulation. This is because a slave device is always active and ready to respond to activity initiated by the master. Once the slave sequence is forked the wishbone reset sequence is started. This causes the wb_drv_bfm to assert and then release reset. Once the reset sequence has completed a series of writes and reads are performed on the wb2spi DUT. The format of the SPI transfer as a memory slave is shown in the table below.

SPI Slave Bus Protocol

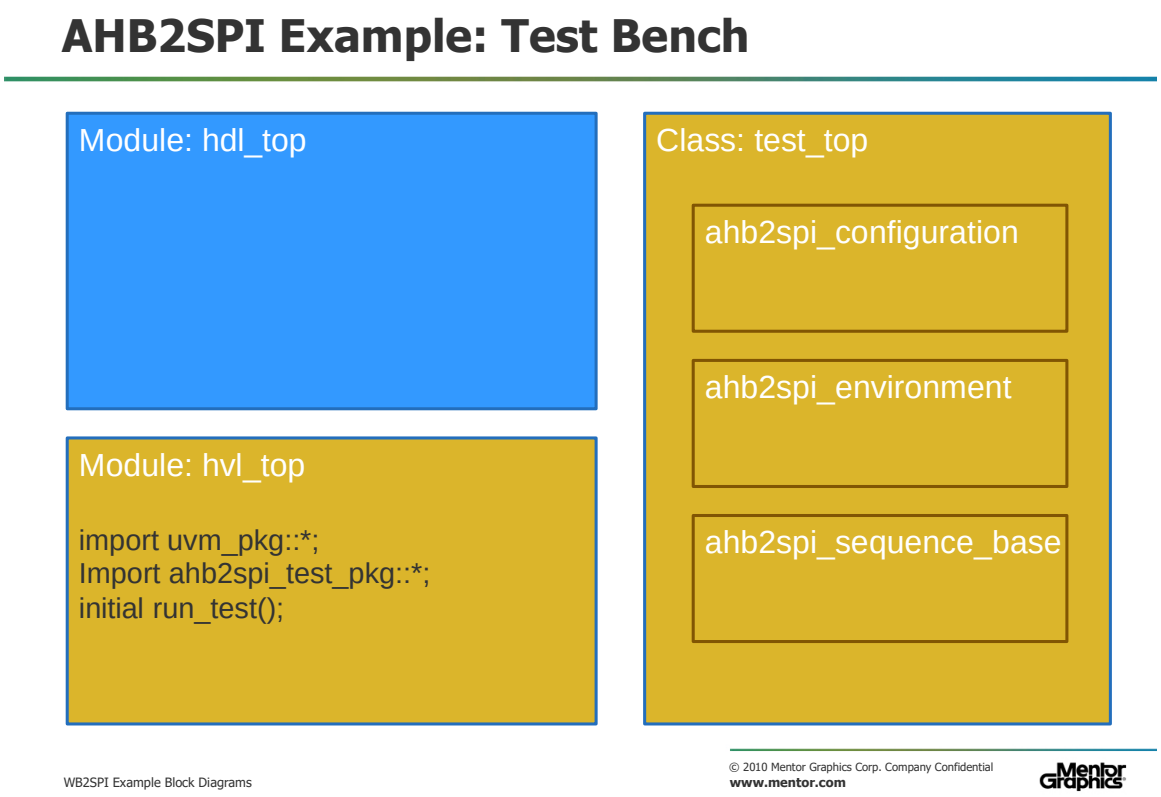
Signal	Data [7]	Data [6:4]	Data [3:0]
MOSI	RW 1: Write, 0: Read	Address[2:0]	Data[3:0]
MISO	STATUS (prev command) 1:Success, 0:Error	Address[2:0]	Data[3:0]

2.1.3 AHB to SPI Example

The AHB2SPI example demonstrates a chip level environment. It is located in the base_examples group. This chip level environment reuses the AHB2WB and WB2SPI block level environments. This environment includes a register model based on the UVM register package. This chip level register model contains a register block for the WB2SPI block level environment.

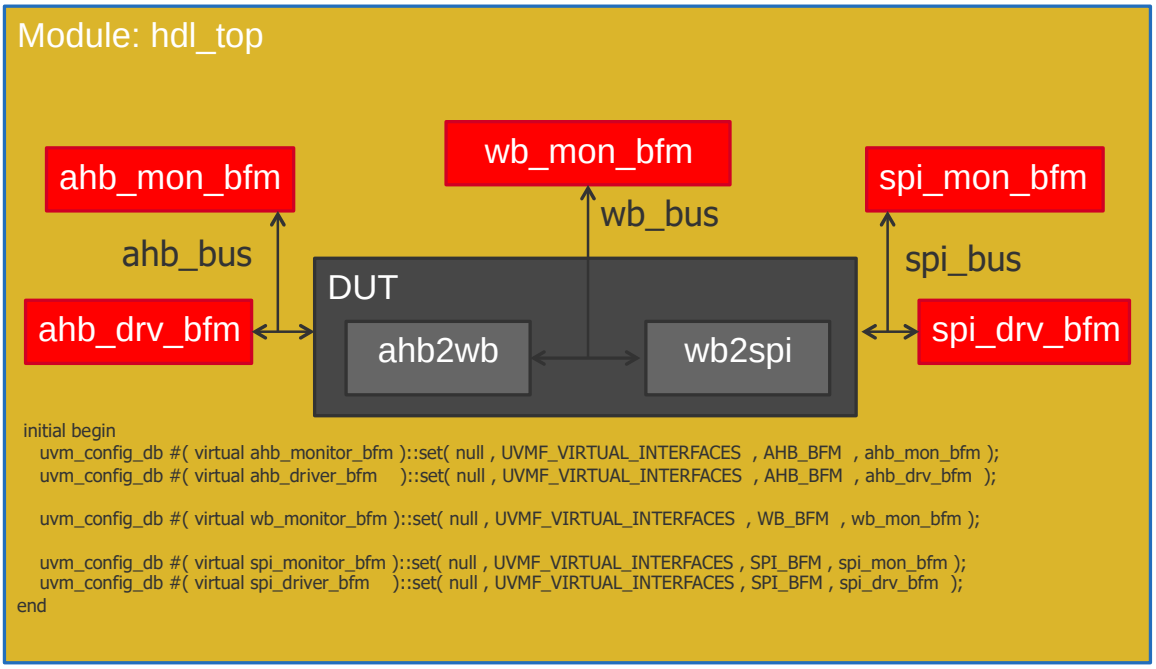
This example demonstrates the following:

1. Chip level environment that reuses block level environments
2. Chip level UVM register model that reuses a block level UVM register model



As with all UVMF test benches, the AHB2SPI test bench is composed of three top levels: hdl_top, hvl_top and test_top. The module named hdl_top contains the DUT, BFMs and signal bundle interfaces that tie them together. All content in hdl_top is synthesizable to support emulation. The module named hvl_top contains all content that must reside within a module but is not synthesizable. This includes importing the test package and calling run_test to start the UVM phases. The class named test_top is the top level UVM test class. It is selected using the +UVM_TESTNAME argument on the command line and constructed by the UVM factory.

AHB2SPI Example: hdl_top

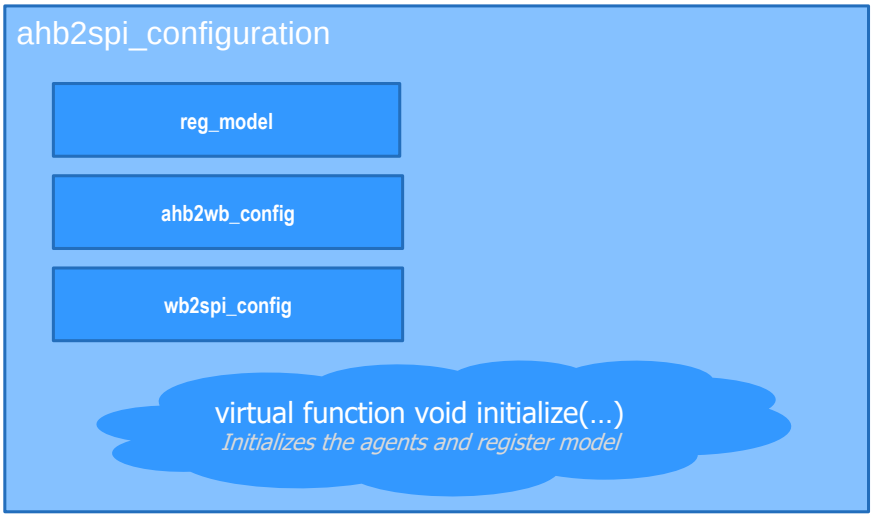


WB2SPI Example Block Diagrams

The hdl_top module contains the DUT and BFMs used to drive and monitor bus activity. The _drv_bfm interfaces provide signal stimulus. The _mon_bfm interfaces observe signal activity and capture signal information for broadcasting to the environment for prediction, scoreboarding and coverage collection. An interface containing all of the signals for the bus ties the monitor BFM, driver BFM and DUT signal ports together. Protocol signals are separated into an interface to enable block to top reuse of environments and monitor BFMs. All BFMs are placed into the uvm_config_db by hdl_top for retrieval by the appropriate agent configuration. This mechanism is described in the section on resource sharing and initialization within UVM Framework.

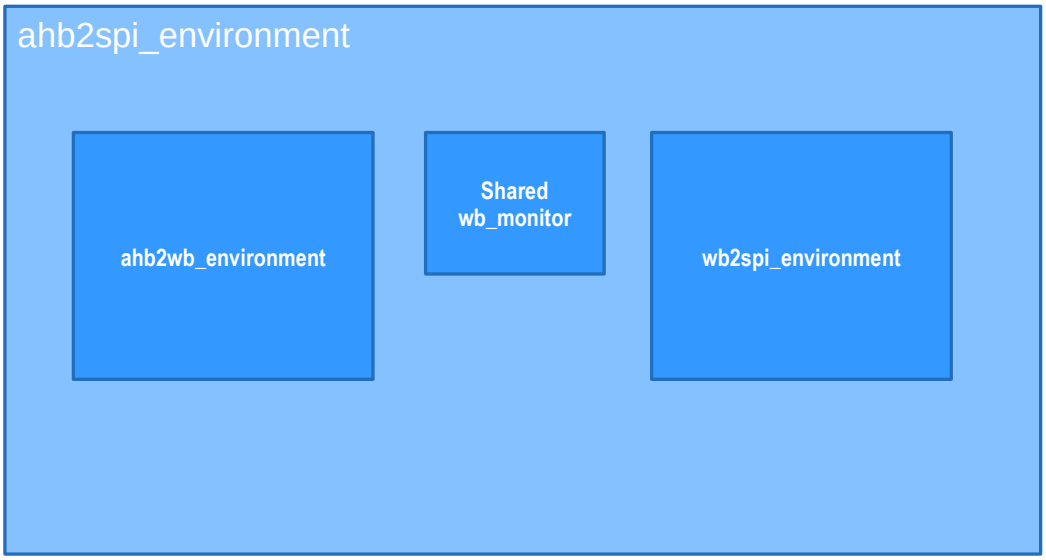
In this example the wishbone bus is internal to the DUT and driven by RTL within the DUT. A wishbone signal bundle interface, wb_bus, is connected to the wishbone bus in the DUT in order to observe bus activity. This can be done using either the SystemVerilog bind construct or hierarchically connecting the signal bundle into the DUT. This wishbone signal bundle is connected to two wishbone monitor BFM. This is to provide the wishbone agent within each of the block level environments a wishbone monitor BFM virtual interface handle. This allows independent prediction, scoreboarding and coverage for each block level environment.

AHB2SPI Example: Configuration



The ahb2spi_configuration class contains a configuration for each block level environment within this chip level environment and a register model. The register model is a UVM register block for the ahb2spi DUT. This register block contains a wb2spi register block for use by the wb2spi block level environment. A function named initialize provides the environment configurations with the simulation level, BLOCK/CHIP, the hierarchical path down to the chip level environment and an array of string names of the interface to be retrieved from the uvm_config_db. The ahb2spi configuration also contains DUT configuration specific variables that can be randomized as needed. The ahb2spi configuration class is constructed, randomized and initialized before the environment build phase is executed. This ensures that the environment can be built according to the configuration for the simulation. Randomization occurs down through the configuration classes. The post_randomize of ahb2spi_configuration randomizes ahb2wb_configuration and wb2spi_configuration applying implication constraints to enforce lower level value options based on upper level values randomly selected.

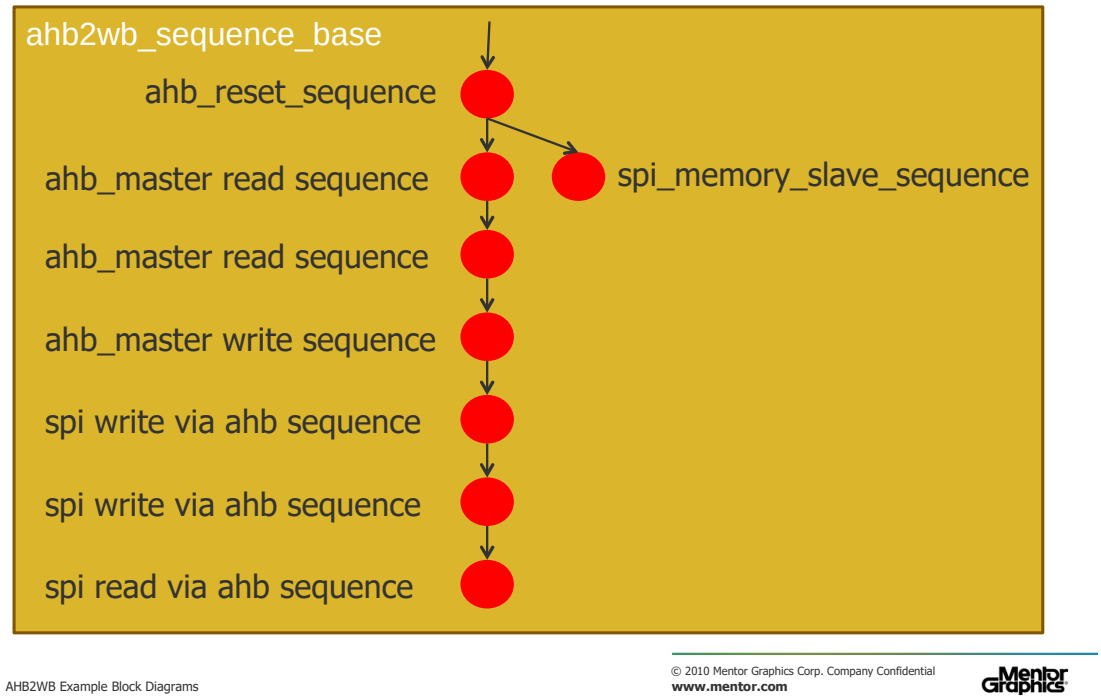
AHB2SPI Example: Environment



WB2SPI Example Block Diagrams

The ahb2spi environment contains the ahb2wb environment and the wb2spi environment. These two block level environments perform the same prediction, scoreboarding and coverage provided when run in the block level bench. Stimulus is driven into the design via the ahb interface. The wishbone interface is now buried in the DUT. It is observed by both environments for prediction, scoreboarding and coverage. Data is sent out through the SPI interface of the DUT. The ahb2wb_environment is configured by the ahb2spi_configuration. The ahb2wb_environment is configured by the ahb2wb_configuration. The wb2spi_environment is configured by the wb2spi_configuration.

The ahb2spi environment creates a wb_monitor to be shared between the two environments that need to observe the wb bus. This wb monitor is constructed by the ahb2spi environment and placed into the uvm_config_db for retrieval by the wb agent within each block level environment. The shared wb_monitor is connected to the single wb_monitor_bfm. WB transactions observed by the wb_monitor_bfm are sent to the shared wb_monitor and broadcasted within each environment.



The top level sequence, named ahb2spi_sequence_base, orchestrates and controls all stimulus within the simulation. The stimulus flow is shown in the diagram above. The first sequence to be started is the ahb reset sequence. This causes the ahb_drv_bfm to assert and then release reset. Once the reset sequence has completed then the SPI memory slave sequence is started. This sequence is forked off because it will remain active throughout the simulation. This is because a slave device is always active and ready to respond to activity initiated by the master. Once the slave sequence is forked a series of writes and reads are performed on the DUT through the ahb port. These operations write and read the SPI memory slave attached to the SPI port of the DUT.

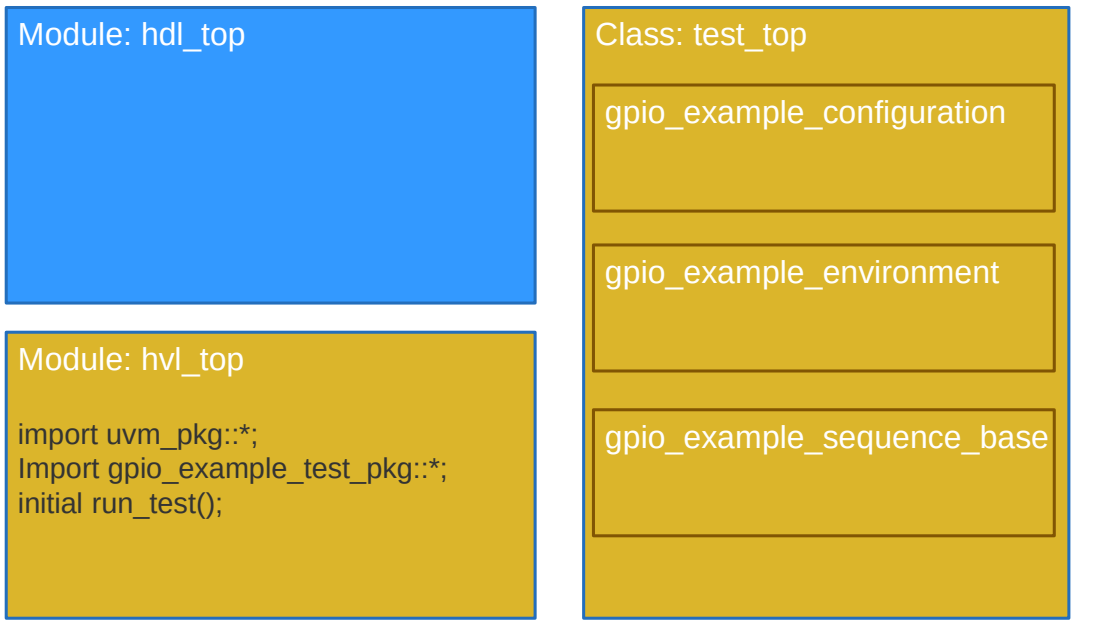
2.1.4 GPIO Example

The GPIO example demonstrates the use of a parameterized interface. The WRITE_PORT_WIDTH and READ_PORT_WIDTH parameters are used to instantiate the BFM interfaces, agent, configuration, and sequence classes. The value for these parameters are defined in the gpio_example_parameters_pkg.

This example demonstrates the following:

1. The creation and instantiation of a parameterized interface

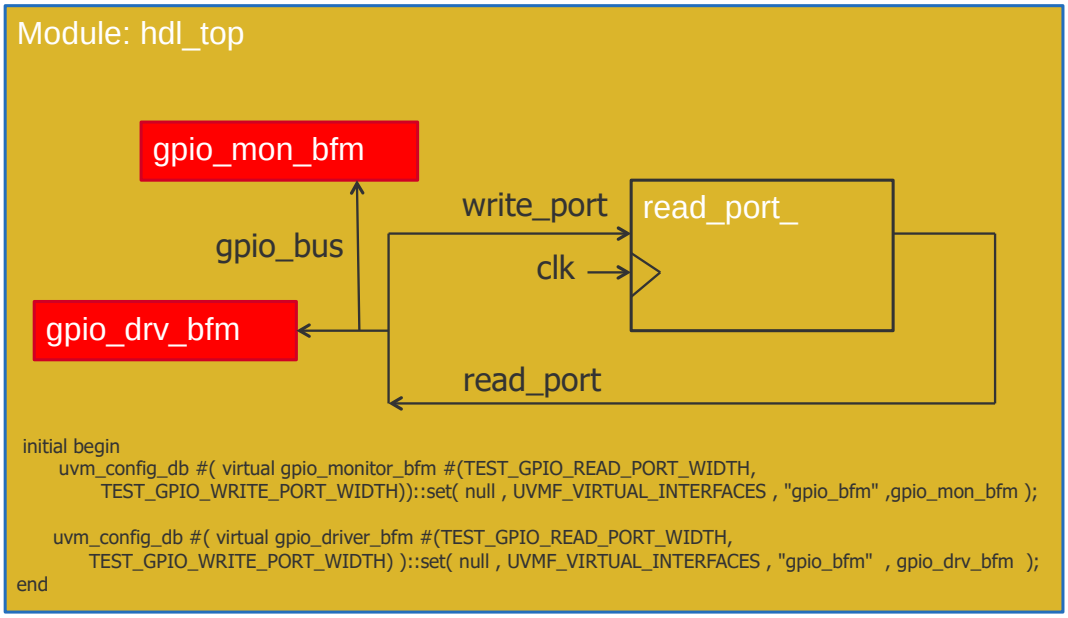
GPIO Example: Test Bench



GPIO Example Block Diagrams

As with all UVMF test benches, the GPIO test bench is composed of three top levels: hdl_top, hvl_top and test_top. The module named hdl_top contains the DUT, BFMs and signal bundle interfaces that tie them together. All content in hdl_top is synthesizable to support emulation. The module named hvl_top contains all content that must reside within a module but is not synthesizable. This includes importing the test package and calling run_test to start the UVM phases. The class named test_top is the top level UVM test class. It is selected using the +UVM_TESTNAME argument on the command line and constructed by the UVM factory.

GPIO Example: hdl_top

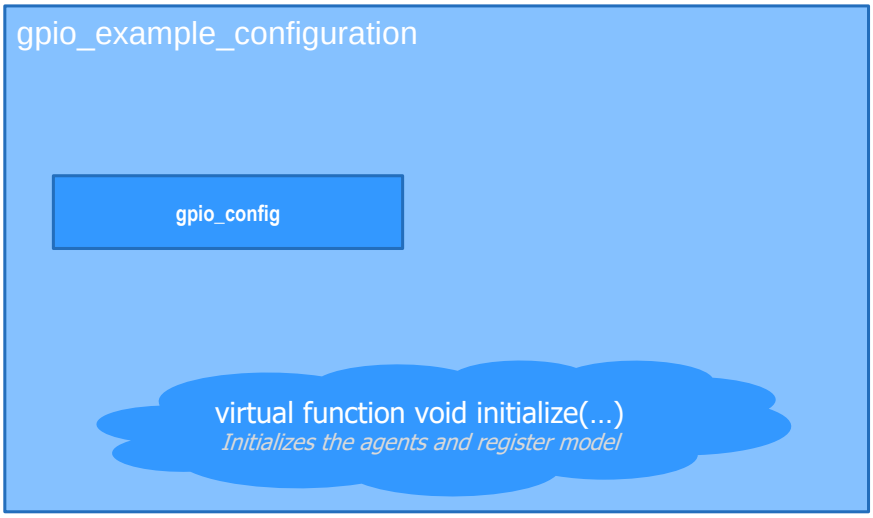


GPIO Example Block Diagrams

The hdl_top module contains the DUT and BFMs used to drive and monitor bus activity. The _drv_bfm interfaces provide signal stimulus. The _mon_bfm interfaces observe signal activity and capture signal information for broadcasting to the environment for prediction, scoreboarding and coverage collection. An interface containing all of the signals for the bus ties the monitor BFM, driver BFM and DUT signal ports together. Protocol signals are separated into an interface to enable block to top reuse of environments and monitor BFMs. All BFMs are placed into the uvm_config_db by hdl_top for retrieval by the appropriate agent configuration. This mechanism is described in the section on resource sharing and initialization within UVM Framework.

In this example the DUT is a simple register named read_port_. The input to the register is the write_port of the gpio_bus. On each clock edge the value on write_port is output on read_port_. The read_port_ value is then assigned to the read_port of the gpio_bus. This inserts a one clock delay between the write_port output and read_port input. This loopback delay is only for demonstration purposes.

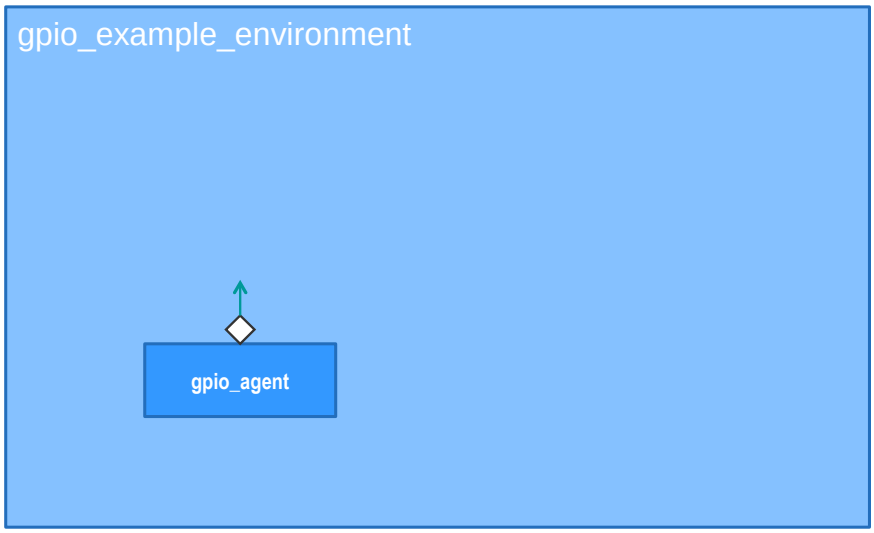
GPIO Example: Configuration



GPIO Example Block Diagrams

The gpio_example_configuration class contains a configuration object for the GPIO agent in the environment. A function named initialize provides the agent configuration with the active/passive state of the agent, the path to its agent in the environment and the string name of the interface to be retrieved from the uvm_config_db.

GPIO Example: Environment



GPIO Example Block Diagrams

The gpio_example_environment only contains the gpio_agent. This is because the purpose of this example is to demonstrate the use of a parameterized interface, agent, configuration and sequence.

GPIO Example: Top Level Sequence

```
58  // ***************************************************************************
59    virtual task body();
60
61       gpio_seq = new("gpio_seq");
62       gpio_seq.start(gpio_sequencer);
63       gpio_config.wait_for_num_clocks(2);  //#20ns;
64       gpio_seq.bus_a = 16'h1234;
65       gpio_seq.bus_b = 16'habcd;
66       `uvm_info("GPIO", gpio_seq.convert2string(), UVM_MEDIUM)
67       gpio_seq.write_gpio();
68       gpio_config.wait_for_num_clocks(2);  //#20ns;
69       gpio_seq.read_gpio();
70       gpio_config.wait_for_num_clocks(2);  //#20ns;
71       `uvm_info("GPIO", gpio_seq.convert2string(), UVM_MEDIUM)
72
73
74    endtask
```

GPIO Example Block Diagrams

The sequence used in this example is different from most sequences in that it is started at the beginning of the simulation and remains throughout the simulation. Writing values to the GPIO write_port and reading values from the GPIO read_port are done through tasks in this sequence. The sequence, named gpio_seq, is an extension to the gpio_sequence located in the gpio_pkg. This extension defines bit assignments to the write_port and read_port. In this case bus_a and bus_b are assigned to the write_port, bus_c and bus_d are assigned from the read_port. The flow of the top level sequence is outlined below:

Initialization:

Line 61: Construct the gpio_seq sequence.

Line 62: Start the gpio_seq sequence. This sequence remains resident throughout the simulation.

Line 63: Wait for two clocks using the wait_for_num_clocks task within the gpio_agents configuration class.

Write operation:

Line 64 and 65: Set the values of bus_a and bus_b variables.

Line 66: Display the variable values in the sequence item within gpio_seq.

Line 67: Write the new values of bus_a and bus_b to the GPIO write_port

Line 68: Wait for two clocks using the wait_for_num_clocks task within the gpio_agents configuration class.

Read operation:

Line 69: Read the values currently on the GPIO write_port and read_port.

Line 70: Wait for two clocks using the wait_for_num_clocks task within the gpio_agents configuration class.

Line 71: Display the variable values in the sequence item within gpio_seq.

2.1.5 Questa VIP Examples

The Questa VIP examples provide UVM Framework environments with instantiations of Questa VIP. They reside in the vip_examples group. These examples can be used to understand where constituent pieces of Questa VIP reside in the environment. They can also be used as a starting point for designs that have standard protocols.

2.1.5.1 AXI4 Example

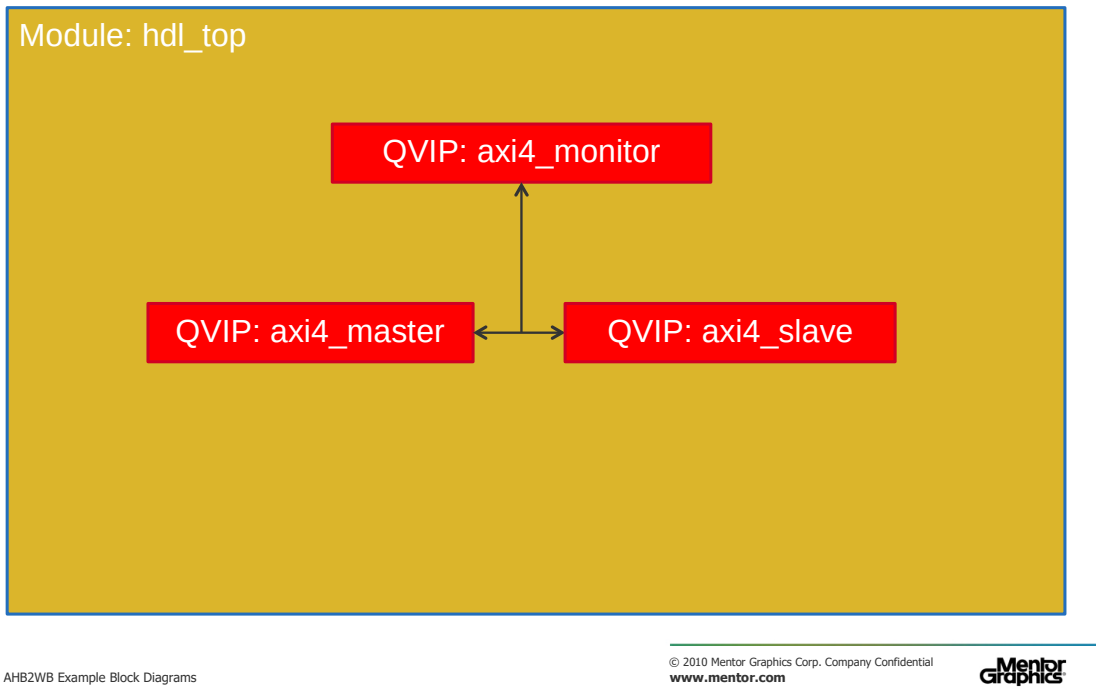
This example demonstrates the various features of the QVIP AXI4 as listed below. This example can be used as a production environment by substituting a design for either axi4_master, axi4_slave or both.

This example demonstrates the following:

1. Block level environment with a QVIP AXI4 master connected to a QVIP AXI4 slave with a QVIP AXI4 monitor observing bus activity.

The table below lists the interfaces and classes used from the QVIP Library and where they are located in the environment.

Component Desicription	**Component Used**	**Location in UVMF**
SystemVerilog interface	Axi4_master Axi4_slave Axi4_monitor	Hdl_top.sv
Configuration	Axi4_vip_config	Vip_axi4_configuration.svh
Agent	Axi4_agent	Vip_axi4_environment.svh
Sequence	Axi4_out_of_order_sequence	Qvip_axi4_bench_sequence_base.svh



2.2 Running UVMF Example Benches

The UVMF examples will run on Windows and Linux. UVMF examples are run from the sim directory located under the *exampleGroup*/project_benches/*benchName* directory. The examples can be run in either command line or GUI mode. The sections below describe how to run the examples in Linux and Windows.

Running the base_examples from within the Questa installation is not recommended. Copy the base_examples folder from the UVMF installation into a scratch area. Run the examples within the scratch area.

2.2.1 Running the Examples in Linux

Makefiles are provided under the sim directory of each example bench for running on Linux. Example benches are under the exampleGroup/project_benches directory. Under each example bench is the sim directory where simulations are run. When in the sim directory do the following to run a simulation:

Set environment variables listed in the Makefiles section of this document.

Run one of the two following commands:

```
make cli
make debug
```

The `cli` make target runs the sim in command line mode. The `debug` make target runs the sim in GUI mode. You can view signals and transactions in debug mode.

Refer to section 3 for details on how to run single simulations via the Makefile structure.

2.2.2 Running the Examples in Windows

The examples as well as template generated code can be run using the makefile on Windows using a windows make utility.

There are also compile.do and run.do scripts provided for running the examples as well as the template generated code in Windows. To use the Tcl scripts set the environment variables listed in the Makefiles section of this document.

2.2.3 Running the Examples on Veloce

By default, the makefiles are configured to run in simulation mode. The following make variables are used to run in emulation. Add these variables to the make command in the format shown as indicated.

This variable is used to run emulation:

USE_VELOCE = 1 (default is 0, i.e. pure simulation mode)

This variable enables emulatable VIP (VTL) use in lieu of pure simulation VIP (QVIP)

USE_VTL = 1 (default is 0, i.e. QVIP mode)

This variable enables the legacy TBX flow instead of the Veloce OS3 emulation flow:

USE_LEGACY_TBX_FLOW = 1 (default is 0, i.e. OS3 flow)

```
make [cli|debug] [USE_VELOCE=1] [USE_VTL=1] [USE_LEGACY_FLOW=1]
```

Note that the combination `USE_VELOCE = 1` and `USE_VTL = 0` is invalid. In the presence of VIP the use of emulation (`USE_VELOCE = 1`) implies the use of emulation-ready VIP (`USE_VTL = 1`). The reverse is not true, i.e. emulation-ready VIP can also be used in pure simulation.

2.2.4 Running the Examples using Questa Verification Run Manager

Questa Verification Run Manager (VRM) is a utility built into Questa that facilitates the execution of regressions in a highly automated way. In addition to being able to build and run simulations, it also easily integrates with grid management software like LSF, automates the process of determining PASS/FAIL for individual runs and automatically manages the collection and merging of coverage data as well as many other capabilities.

The UVMF examples are shipped with a reference VRM control file (RMDB) that illustrates some of the basic VRM capabilities for each bench. A common default.rmdb file resides in `$UVMF_HOME/scripts` which controls VRM and a test list control file resides in each bench's ./sim directory to specify which tests are associated with the given bench. To invoke a VRM regression simply invoke the command "vrun" in a bench's sim directory, as shown:

```
vrun -rmdb $UVMF_HOME/scripts/default.rmdb
```

Refer to section 4 for details on how to invoke UVMF regression test runs with VRM.

2.3 Viewing the Examples using Questa Visualizer

The Mentor Visualizer Debug Environment is Mentor's next generation debugger GUI. If installed and licensed it can be invoked in both post-simulation debug mode or interactively against any of the UVMF test benches through the makefiles.

To run a simulation with Visualizer enabled for the purposes of post-sim debug, set the USE_VIS or USE_VIS_UVM variables to 1 when running "make cli" for a given test. Setting USE_VIS to 1 will log all signals and memories for a given run whereas USE_VIS_UVM will also log SV and UVM constructs. Once a simulation is complete, invoke Visualizer in post-sim debug mode by running "make vis".

To run a simulation using Visualizer as the interactive debugger execute "make debug USE_VIS=1".

Visualizer wave files can also be produced during a VRM-based regression run. See section 4 for details.

3 Makefiles

The UVMF uses a two level makefile structure. Individual packages have a makefile that contains make targets for compiling the package and associated modules. The simulation bench makefile includes the makefiles for all packages used by the bench as well as the uvmf_base_pkg makefile. This gives the bench makefile access to the make targets needed to compile required packages located under verification_ip. The bench makefile also contains make targets for all packages located under the project's tb directory.

3.1 Package Makefile

Each package located under the verification_ip directory has a makefile that contains the make targets required to compile the package. These makefiles are included in the simulation bench makefile depending on which packages under verification_ip are used by the bench. Including each package makefile in the bench makefile allows project bench access to make targets for all packages under verification_ip that are required by the bench.

3.2 Common Makefile

The uvmf_base_pkg makefile is located in the scripts directory. This makefile contains common makefile variables and conditionals used to create commands for compiling and running example UVMF code. The common makefile is included by each project bench makefile.

3.3 Simulation Bench Makefile

Each project bench contains a makefile located in the sim directory under project_benches/<benchName>. This makefile includes the common makefile for compiling required code located under verification_ip. The bench makefile also contains make targets for all packages and modules located under the tb directory.

3.4 User Makefile Variables

3.4.1 Environment variables used for directory structure control

The makefile in verification_ip/scripts contains the following variables that can be set using environment variables. These variables are used to indicate the location of code to be used by UVMF.

UVMF_HOME : This variable points to the home directory of UVMF core code. This represents released, non-user modified, code. This directory should contain the uvmf_base_pkg, common, scripts and base_examples directories. Example: `/tools/Questa/10.7/questasim/examples/UVM_Framework/UVMF_3.6h`

UVMF_VIP_LIBRARY_HOME : This variable points to the directory where reusable UVMF IP is located. It should point to and include the verification_ip directory. Example: `/repository/simulation/reuse/verification_ip`

UVMF_PROJECT_DIR : This variable points to the directory where project benches are located. It should point to and include the project_benches directory. Example: `/projects/simulation/project_name/project_benches`

3.4.2 Command Line Makefile Variables

The makefile in the project_benches/bench/sim directory contains the following make variables that can be set from the command line. These variables have default values that can be seen in the makefile. To change the values of a makefile variable use the following format: `TEST_NAME=my_test`

Variable Name	Description	Default
`TEST_NAME`	Specify which UVM test to execute	`test_top`
`TEST_SEED`	Specify a random seed to use for the simulation	`random`
`USE_INFACT`	Use inFact as part of the simulation run	`0`
`USE_VELOCE`	Use Veloce emulation as part of the simulation run	`0`
`USE_VIS`	Use the Visualizer post-simulation debug environment	`0`
`USE_VIS_UVM`	In addition to RTL debug, enable Visualizer UVM debug capabilities	`0`
`CODE_COVERAGE_ENABLE`	Enable code coverage collection for the given run	`0`
`CODE_COVERAGE_TYPES`	Specify which types of code coverage to collect	`bsf` (Branch, Statement & FSM Coverage)
`CODE_COVERAGE_TARGET`	Specify the target for code coverage collection	`/hdl_top/DUT.`
`EXTRA_VLOG_ARGS`	Specify any extra command-	*Empty*

	line switches and arguments for all vlog commands	
EXTRA_VCOM_ARGS	Specify any extra command-line switches and arguments for all vcom commands	*Empty*
EXTRA_VOPT_ARGS	Specify any extra command-line switches and arguments for the vopt command	*Empty*
EXTRA_VSIM_ARGS	Specify any extra command-line switches and arguments for the vsim command	*Empty*
EXTRA_VRUN_ARGS	Specify any extra command-line switches and arguments for the vrun command	*Empty*
VRUN_ARGS	Default switches to use on vrun command-line when invoking VRM using the 'make vrun' command	-html
RMDB_PATH	Specify the location of the RMDB file to use when invoking VRM using the 'make vrun' command	$UVMF_HOME/scripts/default.rmdb

4 Running Regressions with VRM

4.1 Overview

Questa VRM, or Verification Run Manager, is a key component of the Questa Verification Management suite. With it, a relatively small amount of configuration can describe a highly complex and efficient regression flow that can enable a number of useful capabilities. Configuration of VRM is accomplished through an RMDB file, or "Regression Management DataBase" file. This document focuses mostly on what the UVMF RMDB file has been configured to accomplish and will only touch on basic VRM capabilities when necessary. For detailed information on VRM and RMDB syntax refer to the Verification Run Manager User's Guide in the Questa installation.

The RMDB file for UVMF is centrally located at `$UVMF_HOME/scripts/default.rmdb`. Alongside this is the `default_rmdb.tcl` file which defines a number of Tcl routines that the RMDB file uses to carry out its functions. All UVMF example testbenches as well as any generated UVMF testbenches should point to this default.rmdb file when invoking VRM, and all VRM invocations should be made from a bench's ./sim directory. In other words, there should be no need to create special RMDB files for individual benches.

4.2 Invocation

The simplest way to invoke a VRM regression for a given bench is to use the makefile target "`make vrun`". Information about how to compile the given bench as well as which tests to invoke is pulled from a test list file. The default location and name for the test list

file is "`./sim/testlist`". Information on the content and format of the test list file can be found in a subsequent subsection. By default, the following RMDB behavior is invoked by UVMF:

- Advanced test list file
  - Per-test extra arguments
  - Nested test list files
  - Repeat of tests
  - Control of random seeds on per-test basis
- Parallel build of separate testbenches
- Parallel run of simulations on per-bench basis
- Automatic parallel merge of UCDB output
- Automatic generation of HTML coverage report
- Random test seed generation and control
- Automatic integration of Questa test plan file, if specified

4.2.1 RMDB Controls and Parameters

The following features and capabilities can be controlled and modified by the user in multiple ways. Some capabilities are controlled via RMDB parameters, others through VRM command-line switches. For those controlled via parameter, override with the VRM "-G" switch syntax, e.x. "`vrun -GMASTER_SEED=0`" or alternatively, via the use of a UVMF VRM Initialization Tcl file. See section for details on the format of that file. In the following table, the "Ini Variable Exists" column indicates whether that parameter can be specified via the initialization file. If "Yes", the variable name is the same as the parameter name but all lower-case (i.e. "no_rerun" instead of "NO_RERUN").

Feature	Parameter name	Ini Variable Exists	Default	Notes
Parallelism control	N/A	No	Infinite	Use vrun "-j" switch to reduce the number of allowed parallel processes.
Automatic re-run	NO_RERUN	Yes	Disabled	Enable automatic rerun of failing tests.
Automatic UCDB merge	N/A	No	Enabled	Disable with vrun switch "-noautomerge"
Test list file name	TESTLIST_NAME	No	testlist	Overrides name of test list file but location continues as ./sim
Test list location	TOP_TESTLIST_FILE	No	./sim/testlist	Specify absolute or relative path to test list file
Enable code coverage collection	CODE_COVERAGE_ENABLE	Yes	0	Collect code coverage against DUT
Specify code coverage types	CODE_COVERAGE_TYPES	Yes	bsf	Branch, Statement and FSM code coverage collected by default
Specify code coverage target	CODE_COVERAGE_TARGET	Yes	/hdl_top/DUT.	Recursively collect coverage against the default location for the DUT
HTML Coverage Report Options	HTML_REPORT_ARGS	Yes	-details –source -testdetails -htmldir (%VRUNDIR%)/covhtmlreport	Specify how the HTML coverage report will be generated. Location defaults to the

				sim directory and will include source annotation and coverage details
Streamlined build system (BETA)	USE_VINFO	Yes	0	Run vlog once against a generated .f file instead of multiple vlog invocations.
Generate waves	DUMP_WAVES	Yes	0	Set to 1 to produce Visualizer waveforms for all simulations.
Generate waves on rerun	DUMP_WAVES_ON_RERUN	Yes	0	Set to 1 to produce Visualizer waveforms for simulations that are re-run on failure.
Seed control	MASTER_SEED	No	random	Seeds the top-level random number generator used to seed individual simulations. May be the string “random” or any 32-bit unsigned integer value.
Apply Coverage Exclusions	exclusionfile	Yes	Undefined	If defined, should refer to a valid Tcl file that is intended to be applied to the final merged UCDB prior to coverage report generation. Intended to make “coverage exclude” commands.
Dofile Control	PRE_RUN_DOFILE	Yes	Empty	If non-empty and pointing to a valid file, will be executed before starting any simulation

Dofile Control	USE_TEST_DOFILE	Yes	0	If variable is set and a Tcl file is found matching the UVM test name currently being run, the file will be executed before starting that simulation
Grid System Control	GRIDTYPE	Yes	LSF	Specify which grid system to use. Can be one of the built-in systems or a custom string.
Grid System Control	USE_JOB_MGMT_BUILD	Yes	0	If set, will attempt to invoke build commands into the specified grid system
Grid System Control	USE_JOB_MGMT_RUN	Yes	0	If set, will attempt to invoke run commands into the specified grid system
Grid System Control	GRIDCOMMAND_BUILD	Yes	Empty	Specify the command (bsub, qsub, etc.) with arguments needed to invoke a build into the grid.
Grid System Control	GRIDCOMMAND_RUN	Yes	Empty	Specify the command (bsub,qsub, etc) with arguments needed to invoke a simulation into the grid.
Execution Control	BUILD_EXEC	Yes	Empty	If having difficulty invoking build commands via VRM it is advisable to set this parameter to “exec”
Execution Control	RUN_EXEC	Yes	Empty	If having difficulty

				invoking run commands via VRM it is advisable to set this parameter to “exec”.

4.2.2 UVMF VRM Initialization File

A Tcl-based initialization file can be used to specify many of the parameters described in the table above in a more permanent way. This can be used to override defaults on a user-by-user basis or specify preferences for an entire group.

Use the initialization file by setting the environment variable `$UVMF_VRM_INI` to the full path to the desired file. An example file can be found in `$UVMF_HOME/scripts/uvmf_vrm_ini.tcl` but the file can and should be located elsewhere.

This file, if pointed to, will be sourced prior to running VRM and can be used to both initialize parameters as well as specify Tcl procedures for overriding default VRM behavior or specifying how to invoke non-standard grid management systems.

Parameter overrides take place by specifying a Tcl proc called “`vrmSetup`”. From within this proc use calls to the pre-defined routine “`setIniVar`” to specify the desired value for a given parameter. For example, one can more widely enable support for Visualizer with the following Tcl content:

```
proc vrmSetup {} {
  setIniVar use_vis 1
}
```

Attempts to use `setIniVar` to initialize parameters not in the list above will result in a fatal error. It is legal to have the initialization file source other Tcl files and this mechanism can be used to specify project-level preferences along with user-level preferences.

4.2.3 Test List Format

Test lists provide the RMDB with information about how a given bench is to be built, which bench is associated with a given simulation, and how simulations are to be invoked. Comment lines start with a pound (#). Other non-blank lines must start with a keyword followed by information specific to that keyword

4.2.3.1 TB_INFO Keyword

The TB_INFO keyword specifies how a given testbench should be built. The format for a TB_INFO line is as follows:

```
TB_INFO <bench_name> { <build arguments> } { <runtime arguments> }
```

The build arguments will be applied to the 'make' command that is used to both compile and optimize the testbench environment. Any runtime arguments will be applied to all invocations of vsim for simulations against the given testbench environment.

4.2.3.2 TB Keyword

The TB keyword specifies which test bench should be used for all following tests. The format is as follows:

```
TB <bench_name>
```

After a given TB keyword is found, all subsequent tests will target this testbench. If another TB keyword is used with a different bench name, all subsequent tests after that will utilize the new bench.

4.2.3.3 TEST Keyword

The TEST keyword specifies a test to be executed against the bench that should have been specified earlier with the TB keyword. The format is as follows:

```
TEST <test_name> <repeat_count> <seed0> <seed1> … <seedN> { <per-test arguments> }
```

Only <test_name> is required, all other arguments are optional. If <repeat_count> is omitted a count of 1 is used. A seed for each repeat count may be provided but if any or all are omitted a random seed is generated. The final argument, if provided, will be passed in as additional arguments to vsim for the given test.

4.3 Operation and Results

When VRM is invoked, the test list file is parsed for information on which benches to build and which tests to execute. Once this data model is built, the following occurs:

1. Each test bench is built in parallel
2. After a given test bench has successfully been built, all associated simulation runs are executed in parallel on that bench.
3. UCDB merging takes place as individual simulations finish
4. Once all simulations have finished an HTML coverage report is produced

All operations and output take place under the ./sim/VRMDATA directory. The VRMDATA directory is created when VRM is invoked. Individual bench builds take place in subdirectories, as do individual simulation runs. This allows for a high degree of parallel operation.

- Individual bench compiles take place in `VRMDATA/top/each_top~<bench_name>/build_group/build_task`
- Individual simulations run in `VRMDATA/top/each_top~<bench_name>/build_group/run_fork/run~<bench_name>-<test_name>-<iteration_number>-<random_seed>`

If a build or simulation fails, look in the execScript.log underneath the given directory for information on what went wrong.

5 Using the Python Templates

5.1 Overview

YAML based and Python API based code generators are provided by UVMF for rapid code development. Three code generation templates are provided: interface, environment and bench. The interface template generates the files, infrastructure and interconnect required for an interface package. The environment template generates the files, infrastructure and interconnect required for an environment package. The bench template generates the files, infrastructure and interconnect required for a project bench. The SystemVerilog/UVM code generated by the templates can be simulated as is. This provides a starting point for adding required design and protocol specific code.

A tutorial on using the generators from specification to completed test bench is located in the docs/generator_tutorial directory. This tutorial start with a design specification, walks through the creation of generator input files, to generated code, to post generation modifications to complete the simulation environment.

The templates have been run on the following python releases: 2.6, 2.7. Confirm the python version on machine is at least 2.6 using the following command: python -V

It is important to note that python uses spaces, indentation, to group code blocks. Each line of template configuration files should start at the beginning of the line with no whitespace, indentation.

The input to the generator is a Python configuration script that imports the uvmf_gen.py Python module as well as specifying the work to be done. Example configuration scripts for generating UVMF code using the templates is located under templates/python/api_files.

Use of the YAML based template generator is recommended. It is a more data oriented approach that provides some linting of user written YAML. It also eliminates some of the common mistakes made when using the Python API based template generator. The YAML and Python API template generators are both supported. The YAML based template generator uses the Python API's under the hood for generating SystemVerilog/UVM code. The Python API's are available for customers who prefer a more alogorithmic or programmable mechanism for generating systemVerilog/UVM.

5.2 Installation and Operation

Instructions on installation and operation of the python based UVMF code generators is located in templates/python/templates.README

5.3 Developing an Interface Package using the Python Templates

5.3.1 Format and input required by the template

The UVM Framework includes example interface templates for both YAML and Pythonn API based input. They are located under the templates/python/examples directory. The interface examples are mem_if and pkt_if. The templates require the following information: interface name, signals, transaction variables and configuration variables. The format of the interface template can be viewed in the example. The docs directory contains reference guides for the YAML and Python API template generators.

5.3.2 Steps for using the template

1) Set the environment variables as described above
2) Create the interface template file as described above
3) Execute the interface template
4) Use the list below to add protocol specific code to the new interface package.

5.3.3 Adding protocol specific code

The following list identifies areas where protocol specific code needs to be added to the new interface package. The following files listed assume the interface generated is named abc_pkg. The UVMF_CHANGE_ME string can be used to identify locations for code addition or changes within various files.

1) src/pkt_driver_bfm.sv: Implement protocol driving in the provided do_write task. The entry point into the driver BFM is the access task. If needed, replace the do_write task with the task needed and call this task from the access task. Example tasks include do_read, do_burst, do_transfer, etc.
2) src/pkt_monitor_bfm.sv: Implement protocol monitoring in the provided do_monitor task.
3) Add new sequences as needed. All new sequences should be extended from pkt_sequence_base.

5.3.4 Data flow within generated interface

The interface generated by the UVMF code generators actively drives and monitors data as generated. The driver automatically requests a transaction from the sequencer, waits a few clocks, then requests another transaction. The monitor automatically broadcasts default values every few clocks. The do_transfer task within the driver BFM must be completed in order to see pin activity on the bus. The do_monitor task within the monitor BFM must be completed in order to broadcast observed signal values. The sections below outline the flow of data within the generated interface components.

5.3.5 Data flow within generated monitor

The generated interface components associated with monitoring and broadcasting observed signal activity are operational as generated. The flow outlined below describes the setup, enabling, and operation of this flow. This flow is described for an interface protocol named mem.

1. Mem_monitor.svh: set_bfm_proxy_handle() is called in uvmf_monitor_base::connect_phase() to place a handle to this UVM based monitor within the mem_monitor_bfm. This handle is used by mem_monitor_bfm to send observed data to mem_monitor for broadcasting through the agents analysis_port. This is a setup activity automatically performed once in the connect phase.
2. Mem_monitor.svh: start_monitoring() task called in run_phase() to enable signal monitoring within mem_monitor_bfm. This is an enabling activity automatically performed once in the run_phase.
3. Mem_monitor_bfm.sv: Once monitoring is enabled a forever loop is entered that performs the following two steps

a. Do_monitor(): This task observes and captures signal values according to the protocol. Observed values are returned through the task arguments.
b. Proxy.notify_transaction(): This function takes values from the do_monitor() task and sends them to mem_monitor for broadcasting in the UVM environment. The proxy variable is a handle to mem_monitor initialized in step 1. Since notify_transaction is a function it returns in zero time allowing do_monitor to immediately return to observing signal activity.

4. Mem_monitor.svh: The notify_transaction function performs the following steps when called:
   a. Construct a transaction for broadcasting
   b. Set values in the transaction based on values received as arguments to notify_transaction.
   c. Add the transaction to the waveform transaction viewing stream if the transaction_viewing_stream flag is on in the agent configuration
   d. Broadcast the transaction out of the agent analysis_port named monitored_ap.

5.3.6 Data flow within generated driver

The generated interface components associated with driving signal activity are operational as generated. The flow outlined below describes the operation of this flow. This flow is described for an interface protocol named mem. The flow for an agent acting as an initiator is different than the flow for a responder agent. Each of these flows are described below.

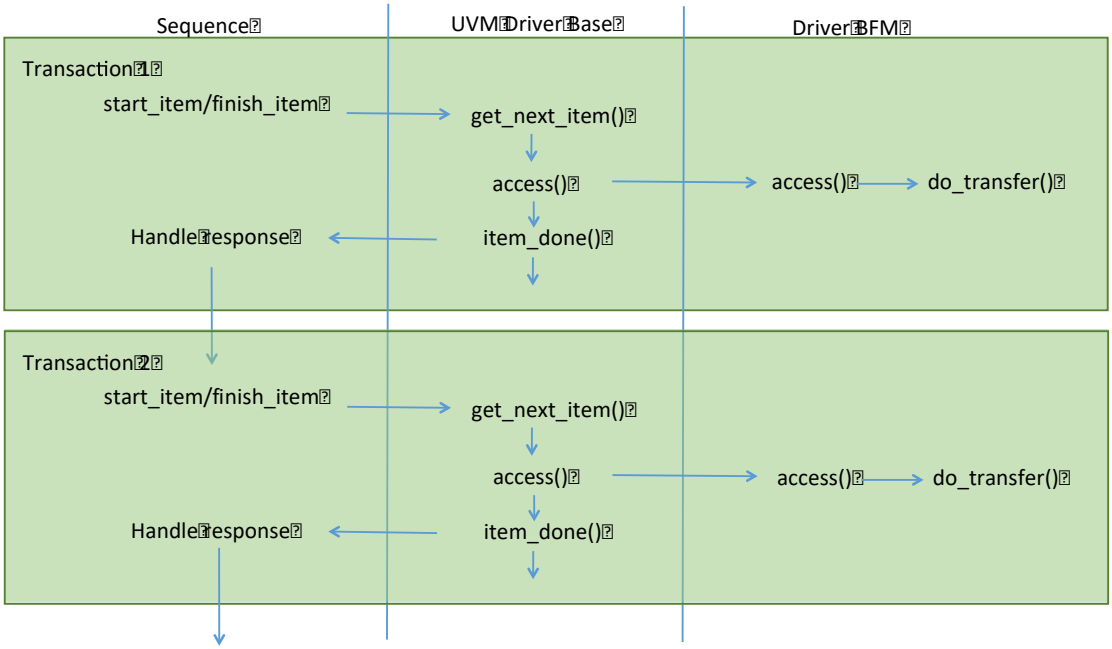
5.3.6.1 Driver flow for initiator

An initiator, or master, agent initiates the transfer of information. The information for this transfer is received from a sequence item. The agent requests this information from a sequence according to the protocol timing. The flow listed below includes steps performed at the sequence.

1. Mem_random_sequence.svh: Executing the start() task of this sequence executes the body() task within this sequence. The body() task performs the following steps:
   a. Construct a mem_transaction named req.
   b. Execute start_item(req) to indicate to sequencer in mem agent that a transaction is available for the mem driver. This call blocks until the mem_driver requests a transaction from the sequencer through the get_next_item() task.
2. Mem_driver.svh: The first of three operations is performed in a forever loop within the run_phase(). This first task is the get_next_item() task which gets a transaction from the sequence via the sequencer. When the driver calls get_next_item() the start_item() in the sequence unblocks. The sequence then randomizes the transaction and calls finish_item(). Finish_item() blocks until the driver calls item_done().

3. Mem_driver.svh: The second of three operations is performed in a forever loop within the run_phase(). This second task is the access() task which takes the information from the transaction and passes it to the mem_driver_bfm. The access task is a standard task used to communicate between the mem_driver and mem_driver_bfm. All UVMF based interface agents use this task to communicate to the BFM regardless of protocol details.
4. Mem_driver_bfm.sv: The access() task receives transaction variables and calls tasks that perform protocol specific activity. The generated task is called do_transfer(). This is a generic name and can be changed to do_write, do_read, do_burst, etc. to perform protocol specific operations. These tasks take transaction variables passed in through the task arguments and drives signal values to implement the protocol. All do_transfer() task arguments are listed as inputs. An arguments direction can be changed to output in order to send observed data back to the calling sequence. Be sure to change the same argument to output in the access() task. This will result in the value captured on the bus being sent to the sequence that initiated the transfer.
5. Mem_driver.svh: The third of three operations is performed in a forever loop within the run_phase(). This third task is the item_done() task which indicates to the sequencer that the driver is done with this transaction. If the transaction is to be returned to the sequence then the txn is passed back to the seuqence using item_done(txn).
6. Mem_random_sequence.svh: When the mem_driver calls item_done() the finish_item() in the sequence unblocks. If the driver is returning a transaction using item_done(txn), which is enabled using the return_transaction_response flag in the agent configuration, then the sequence must perform a get() to receive the response transaction from the driver.

The diagram below illustrates the driver flow within an initiator.



5.3.6.2 Driver flow for responder

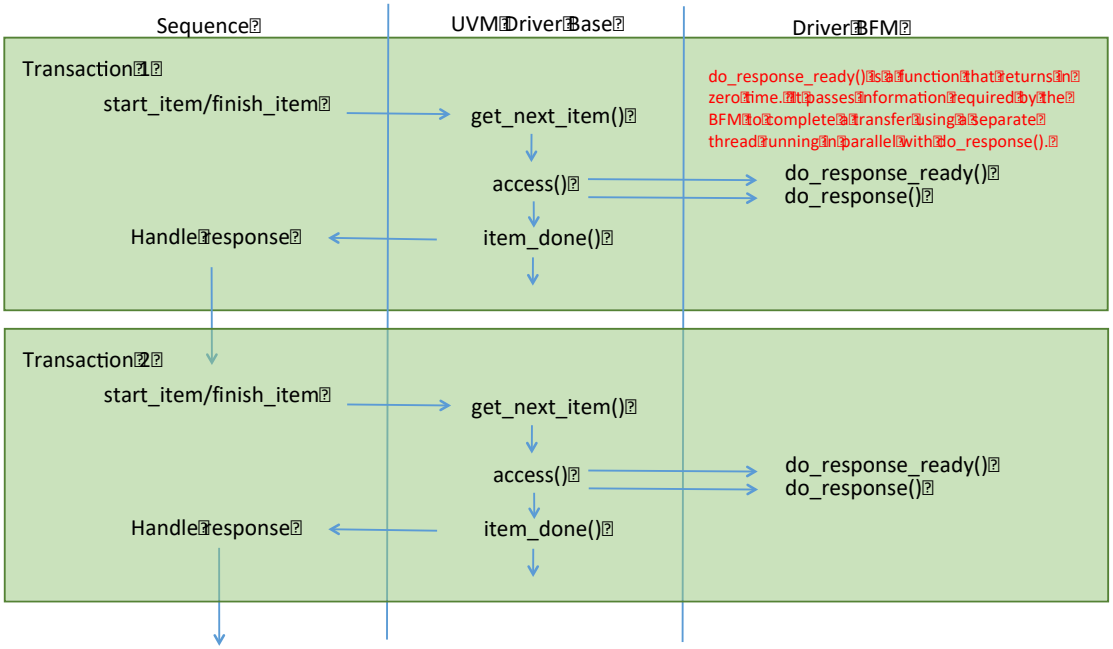
A responder, or slave, agent responds to a transfer started by an initiator or master. Once a transfer is initiated the responder agent sends information about the transfer to the sequence. The sequence returns information needed by the responder agent to complete the transfer. The flow listed below includes steps performed at the sequence.

1. Mem_responder_sequence.svh: Executing the start() task of this sequence executes the body() task within this sequence. The body() task performs a forever loop that does the following steps:
    a. Construct a mem_transaction named req.
    b. Execute start_item(req) to indicate to sequencer in mem agent that a transaction is available for the mem driver. This call blocks until the mem_driver requests a transaction from the sequencer through the get_next_item() task.
2. Mem_driver.svh: The first of three operations is performed in a forever loop within the run_phase(). This first task is the get_next_item() task which gets a transaction from the sequence via the sequencer. When the driver calls get_next_item() the start_item() in the sequence unblocks. The sequence then randomizes the transaction and calls finish_item(). Finish_item() blocks until the driver calls item_done().
3. Mem_driver.svh: The second of three operations is performed in a forever loop within the run_phase(). This second task is the access() task. The access task is a standard task used to communicate between the mem_driver and mem_driver_bfm. If the agent is configured as a RESPONDER then the access()

task calls the do_response_ready() to pass variables required by mem_driver_bfm to complete a transfer. If this is the first call to do_response_ready() then the BFM will do nothing since there is not currently an active transfer in place. The do_response_ready() is a function which returns in zero time. This allows the mem_driver_bfm to complete the current transfer while waiting for the next transfer to be initiated. This is to support phased, pipelined, bus protocols. Once do_response_ready() returns the access() task calls the response() task within mem_driver_bfm.

4. Mem_driver_bfm.sv: The response() task blocks until another transfer is started by the initiator/master. Once a transfer is initiated this task returns with information about the initiated transfer. This information would include address, read/write indication, burst size, etc. This information is returned to the sequence. The sequence takes this information and responds with the information required by mem_driver_bfm in order to complete the transfer.
5. Mem_driver.svh: The third of three operations is performed in a forever loop within the run_phase(). This third task is the item_done() task which indicates to the sequencer that the driver is done with this transaction. If the transaction is to be returned to the sequence then the txn is passed back to the seuqence using item_done(txn).
6. Mem_responder_sequence.svh: When the mem_driver calls item_done() the finish_item() in the forever loop in this sequence unblocks. If the driver is returning a transaction using item_done(txn), which is enabled using the return_transaction_response flag in the agent configuration, then the sequence must perform a get() to receive the response transaction from the driver. Since the sequence and mem_driver share the transaction handle the sequence can access the data in the transfer within the req handle.

The diagram below illustrates the driver flow within a responder including the initial setup.



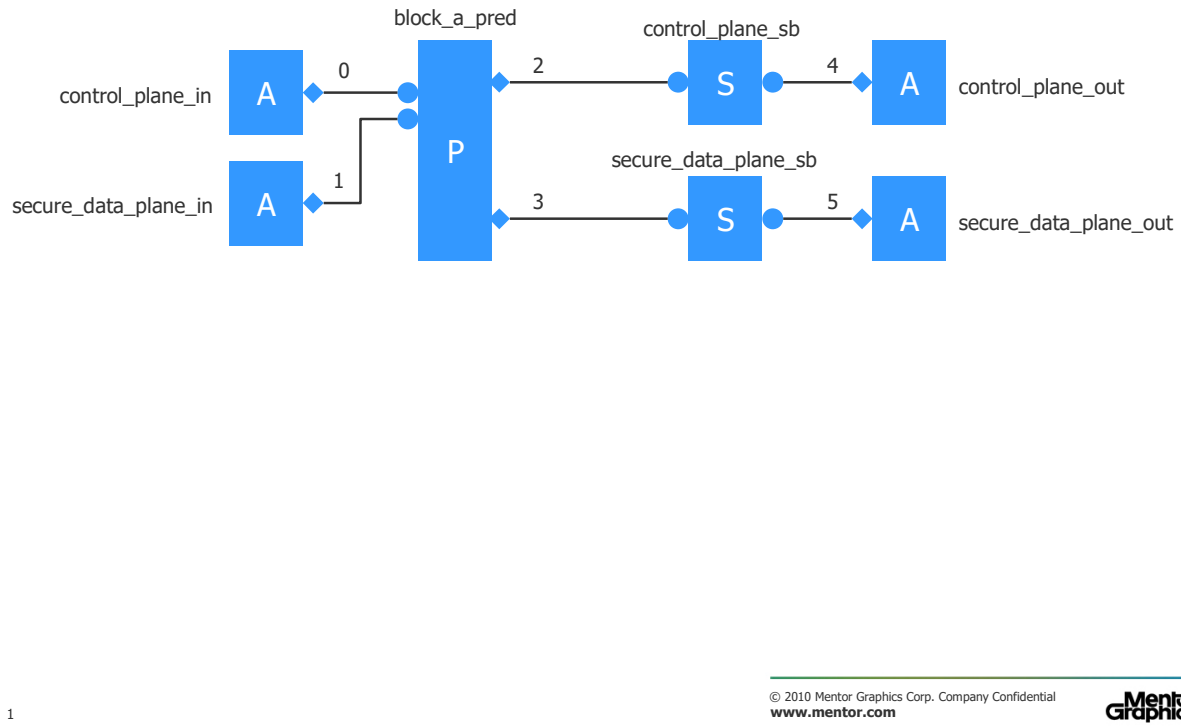
5.4 Developing an Environment Package using the Python Templates

5.4.1 Format and input required by the template

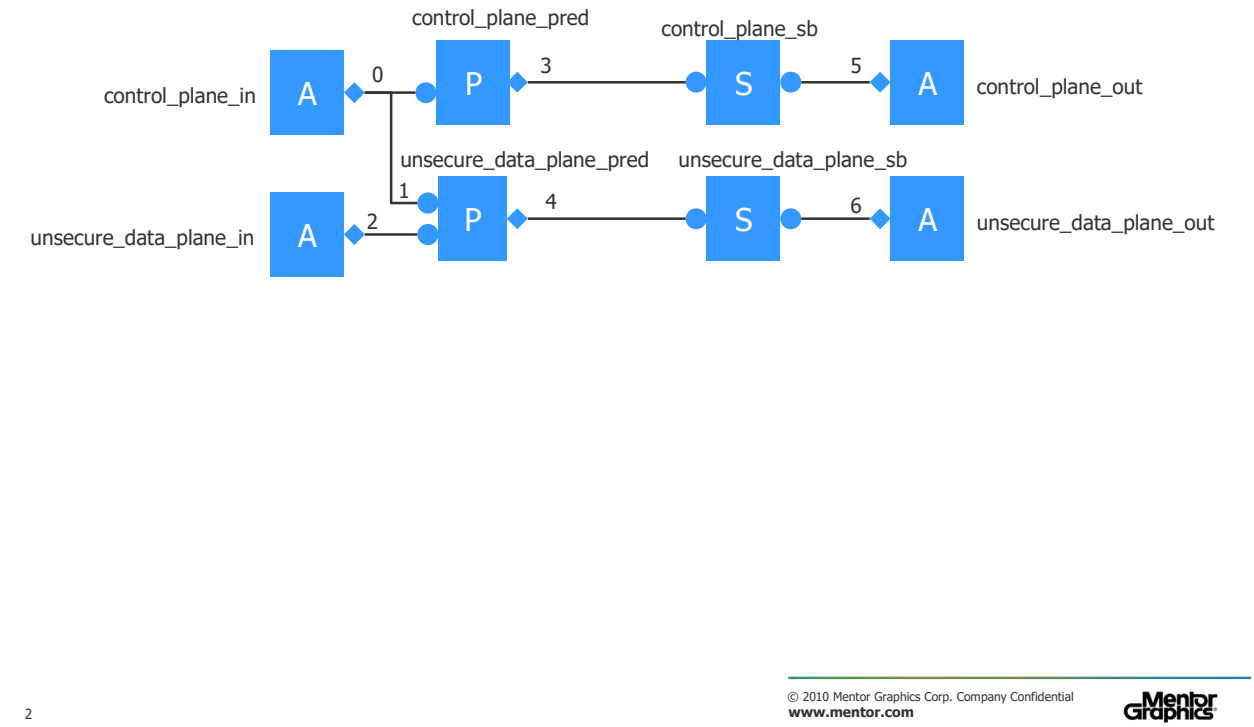
The UVM Framework includes example environment templates. They are located under the templates/python/examples directory. The names of the example environment templates are block_a_env, block_b_env, chip_env, and block_c_env. The template requires the following information: environment name, agents contained in the environment, analysis components contained in the environment, scoreboards contained in the environment and the connections between agents, analysis components and scoreboards. The format can be viewed in the examples. The docs directory contains reference guides for the YAML and Python API template generators.

Block_a is an example of an environment and bench without parameters. Block_b is an example of an environment and bench that uses parameters. Chip_env is an example of a chip level environment that intantiates a block_a sub environment and a block_b sub environment. The block_c example demonstrates use of Questa VIP for standard protocols in an environment that also contains custom protocols, predictors, and scoreboards.

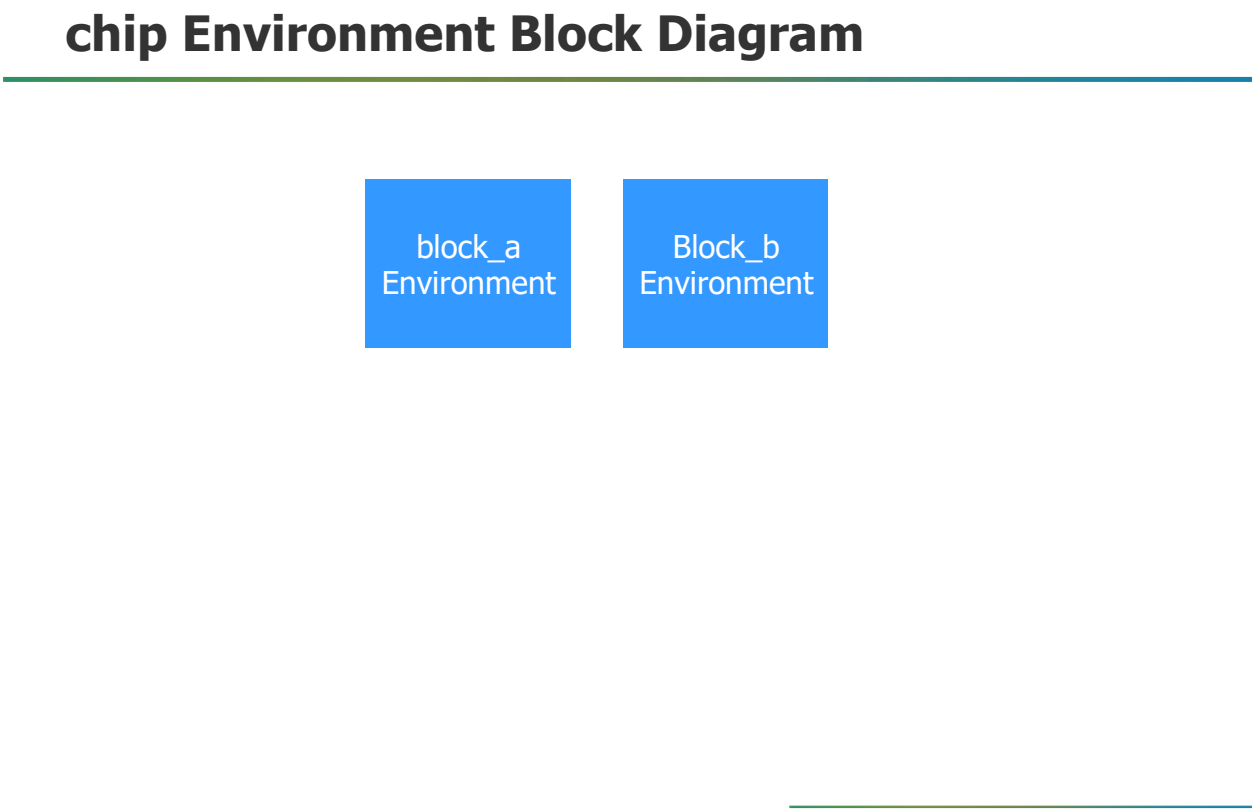
5.4.2 Block diagram of the block_a environment example block_a_env_config.py



5.4.3 Block diagram of the block_b environment example block_b_env_config.py



5.4.4 Block diagram of the chip level environment example: chip_env_config.py



5.4.5 Steps for using the template

1) Set the environment variables as described above
2) Create the environment template file as described above
3) Execute the environment template
4) Use the list below to add DUT specific code to the new environment package.

5.4.6 Adding DUT specific code

The following list identifies areas where DUT specific code needs to be added to the new environment package. The following files listed assumes the environment generated is named abc_prj_env_pkg. The UVMF_CHANGE_ME string can be used to identify locations for code addition or changes within various files.

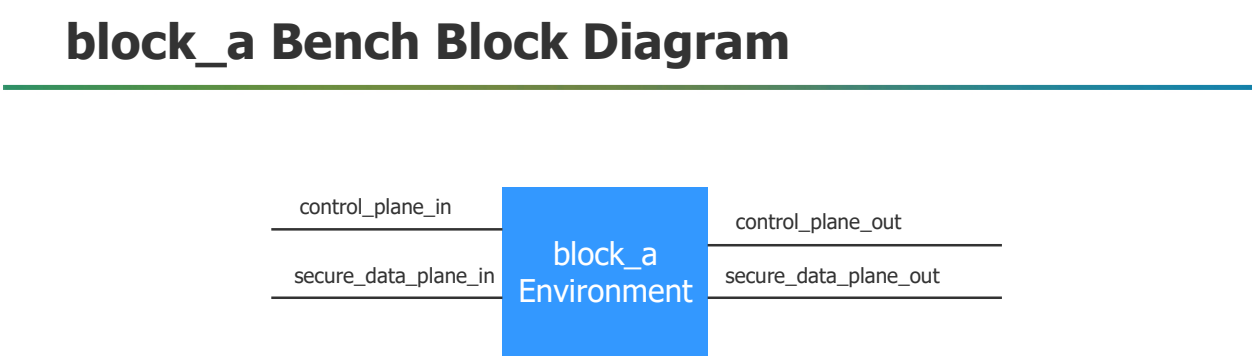
1) src/block_a_env_configuration.svh: Configure agents as required by the DUT.
2) Implement the prediction model in the write…ap function within the class created using the defineAnalysisComponent API.
3) Add new sequences as needed in the src directory. All new sequences should be extended from block_a_sequence_base. Be sure to add new sequences to the environment package: block_a_env_pkg.sv

5.5 Developing a Project Bench using the Python Templates

5.5.1 Format and input required by the template

The UVM Framework includes example project bench templates. They are located under the templates/python/examples directory. The names of the example bench template are block_a_bench, block_b_bench, chip_bench, and block_c_bench. The template requires the following information: bench name and agents contained in the environment used in the bench. The format of the bench template is shown in the examples. The docs directory contains reference manuals for the YAML and Python API based generators.

5.5.2 Block diagram of the block_a block level bench example: block_a_bench_config.py



5.5.3 Steps for using the template

1) Set the environment variables as described above
2) Create the bench template file as described above. When using the Python API based generator, it is important to note that the order of BFM's in the bench template must match the order of agents in the environment template for the environment this bench will instantiate. This is because the ACTIVE/PASSIVE state of each agent is passed as an array to the agents within environments and sub-

environments. These agents receive their ACTIVE/PASSIVE state based on their order in the environment generation file.

3) Execute the bench template
4) Use the list below to add DUT specific code to the new bench.

5.5.4 Adding DUT specific code

The following list identifies areas where DUT specific code needs to be added to the new bench. The following files listed assumes the bench generated is named abc_ben. The UVMF_CHANGE_ME string can be used to identify locations for code addition or changes within various files.

1) Block_a_ben/tb/test bench/hdl_top.sv: Instantiate the DUT and connect ports to the signals in the interface busses, _bus.
2) Change the clock, clk, half period and reset, rst, assertion duration according to testing needs.
3) Add new sequences as needed in the block_a_ben/tb/sequences/src directory. All new sequences should be extended from block_a_ben_bench_sequence_base. Be sure to add new sequences to the environment package: tb/sequences/block_a_ben_sequence_pkg.sv

5.5.5 Template generated interface documentation

The bench generation template creates a file that documents all interfaces in the bench. This file is in csv format and can be imported by Excel to create an interface table. The file is located in docs/interfaces.csv. The interface table created from running block_a_env_config.py is shown below.

Interface Description	Interface Type	Interface Transaction	Interface Name
control_plane_in	mem_driver_bfm mem_monitor_bfm	mem_transaction	mem_pkg_control_plane_in_BFM
control_plane_out	mem_driver_bfm mem_monitor_bfm	mem_transaction	mem_pkg_control_plane_out_BFM
secure_data_plane_in	pkt_driver_bfm pkt_monitor_bfm	pkt_transaction	pkt_pkg_secure_data_plane_in_BFM
secure_data_plane_out	pkt_driver_bfm pkt_monitor_bfm	pkt_transaction	pkt_pkg_secure_data_plane_out_BFM

5.5.6 Generated clock and reset drivers

When the bench is generated with the Veloce Ready flag set, ben.veloceReady = True, the bench clock and reset will be driven using an emulation ready clock and reset generator. This generator is described in the UVM_Co-Emulation_Utility_Library_User_Guide document located in the docs directory. This clock and reset is used to drive the clock and reset signals within each interface BFM.

When the bench is generated with the Veloce Ready flag clear, ben.veloceReady = False, the bench clock will be driven using a simple forever loop that inverts the current value of the clock. The reset, rst, has a default value of 0 and is not changed within the generated bench. The clock, clk, half period and the assertion of reset must be modified by the user according to the testing requirements.

5.5.7 BFM and Agent Ordering

Important note about generating a chip level bench for an environment that contains sub environments when using the Python API based generator:

The list of BFM's in the chip level bench must match the order of agents within the top level environment config file and any sub environment config files. For example, the list of BFM's within chip_bench_config.py matches the order of agents within both block_a and block_b environment configuration files:

```
## block a environment agents in the same order listed in block a config file
ben.addBfm(
        'control_plane_in',      'mem',
        'clock', 'reset','ACTIVE',
        {},
        'environment/block_a_env')
ben.addBfm(
        'internal_control_plane_out',     'mem',
        'clock', 'reset','PASSIVE',
        {},
        'environment/block_a_env')
ben.addBfm(
        'secure_data_plane_in',
        'pkt', 'pclk', 'prst','ACTIVE',
        {},
        'environment/block_a_env')
ben.addBfm(
        'secure_data_plane_out', 'pkt',
        'pclk', 'prst', 'ACTIVE',
        {},
        'environment/block_a_env')

## block b environment agents in the same order listed in block b config file
ben.addBfm(
        'internal_control_plane_in',       'mem',
        'clock','reset','PASSIVE',
        {'ADDR_WIDTH':'TEST_CP_IN_ADDR_WIDTH','DATA_WIDTH':'TEST_CP_IN_DATA_WIDTH'},
        'environment/block_b_env')
ben.addBfm(
        'control_plane_out',      'mem',
        'clock','reset','ACTIVE',
        {'ADDR_WIDTH':'TEST_CP_OUT_ADDR_WIDTH'},
        'environment/block_b_env')
ben.addBfm(
        'unsecure_data_plane_in', 'pkt',
        'pclk','prst','ACTIVE',
        {'DATA_WIDTH':'TEST_UDP_DATA_WIDTH'},
        'environment/block_b_env')
ben.addBfm(
        'unsecure_data_plane_out','pkt',
```

```
'pclk', 'prst','ACTIVE',
{},
'environment/block_b_env')
```

Note the additional argument to the addBfm function. This identifies the path to the sub environment and is used to create the initial wave.do for viewing transactions from sub environments in the wave window. The default value for this field is the default value for top level environments, 'environment'.

5.5.8 Connecting to internal DUT ports

Interfaces that are internal only and as a result are not primary I/O to the bench need to be connected manually. In the chip_bench example the interfaces to be connected are the internal_control_plane_in and internal_control_plane_out busses. The generated code looks like this in hdl_top.sv:

```
mem_if      internal_control_plane_out_bus(.clock(clk), .reset(rst));
mem_if      internal_control_plane_in_bus(.clock(clk), .reset(rst));

mem_monitor_bfm internal_control_plane_out_mon_bfm(internal_control_plane_out_bus);
mem_monitor_bfm internal_control_plane_in_mon_bfm(internal_control_plane_in_bus);
```

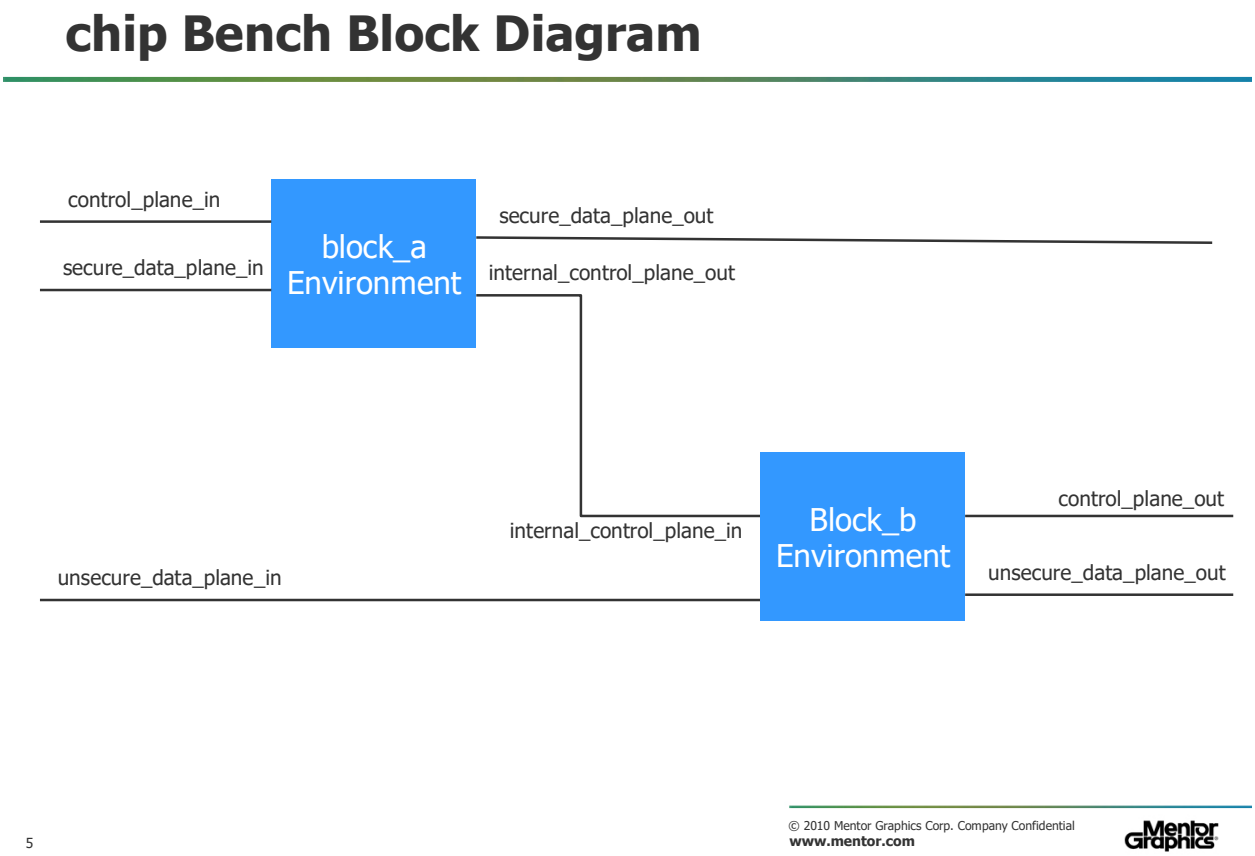
This code must be converted to the following to connect the internal interfaces between the two block levels. The two signal bundles of type abc_if are combined into one named internal_control_plane_bus. The two abc_monitor_bfm's are connected to the new signal bundle, internal_control_plane_bus.

```
mem_if      internal_control_plane_bus(.clock(clk), .reset(rst));

mem_monitor_bfm internal_control_plane_out_mon_bfm(internal_control_plane_bus);
mem_monitor_bfm internal_control_plane_in_mon_bfm(internal_control_plane_bus);
```

Further optimization can be achieved by having the two environments share the same UVM based mem_monitor using the mechanism shown in the ahb2spi environment example.

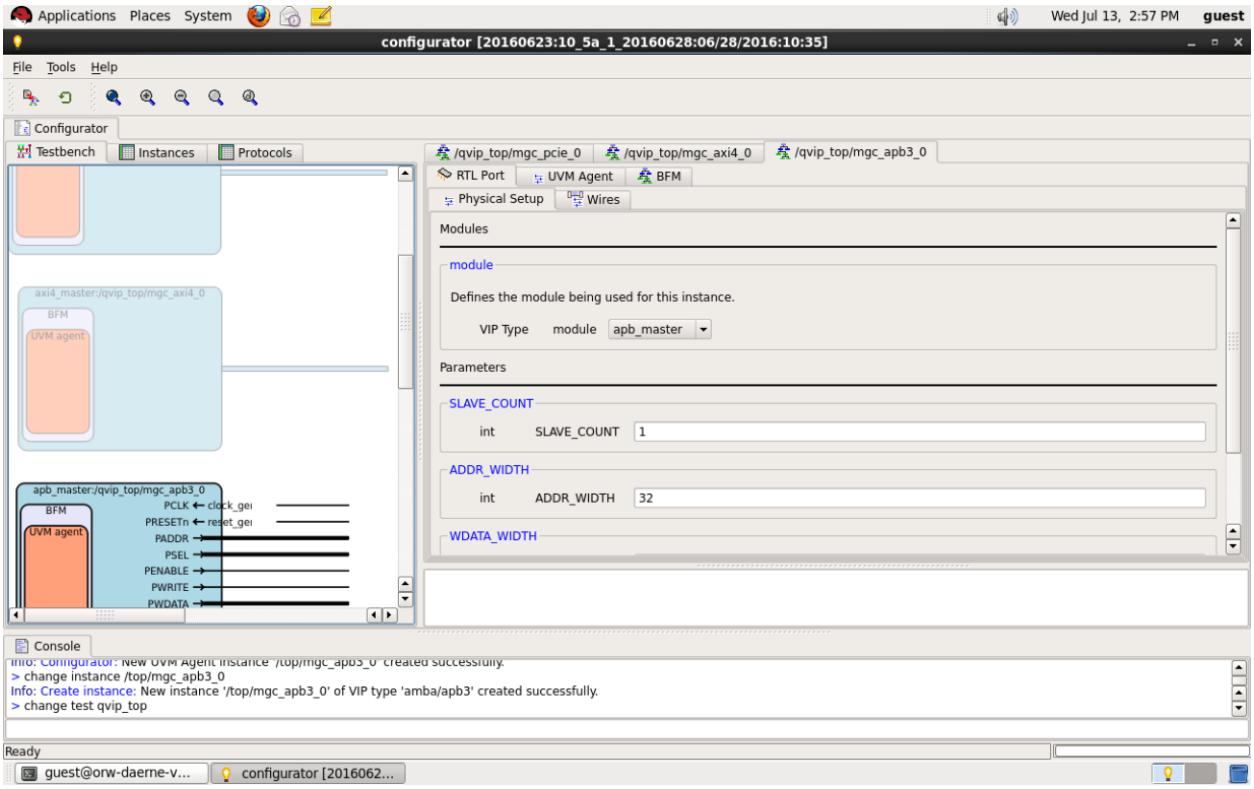
5.5.9 Block diagram of the chip level bench example: chip_bench_config.py



5.6 Using the Questa VIP Configurator in tandem with UVMF code generators

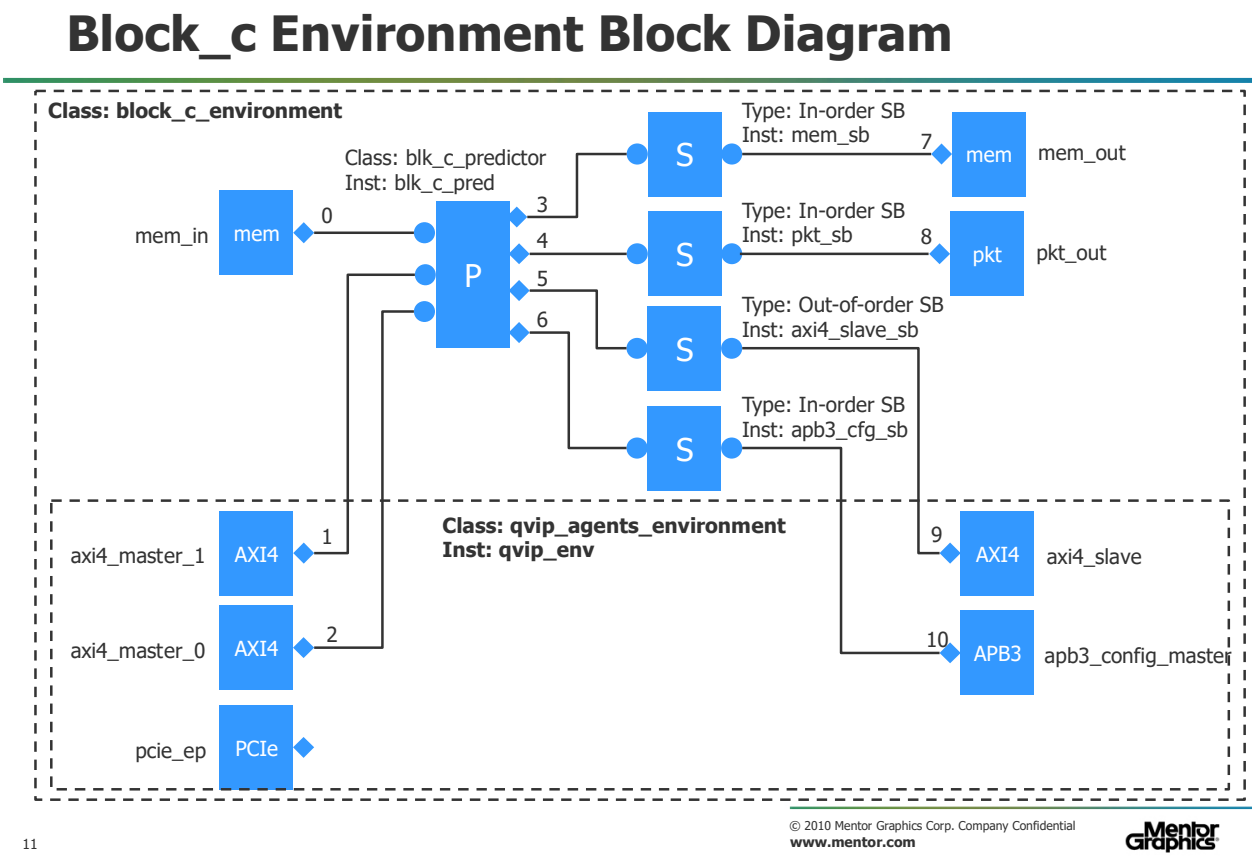
The Questa VIP Configurator is used to generate UVMF based environments containing standard protocols. The UVMF environment generated by the configurator contain the agents and configuration policies for each standard protocol selected by the user. This environment can be automatically included in a UVMF environment containing custom protocol agents, predictors, coverage components, scoreboards, etc.

Documentation on the QVIP Configurator can be found in the QVIP installation. A snapshot of the QVIP Configurator is shown below


Applications Places System
Wed Jul 13, 2:57 PM guest
configurator [20160623:10_5a_1_20160628:06/28/2016:10:35]
File Tools Help
Configurator
Testbench
Instances
Protocols
/qvip_top/mgc_pcie_0
/qvip_top/mgc_axi4_0
/qvip_top/mgc_apb3_0
RTL Port
UVM Agent
BFM
Physical Setup
Wires
Modules
module
Defines the module being used for this instance.
VIP Type module apb_master
Parameters
SLAVE_COUNT
int SLAVE_COUNT 1
ADDR_WIDTH
int ADDR_WIDTH 32
WDATA_WIDTH
axi4_master:/qvip_top/mgc_axi4_0
BFM
UVM agent
apb_master:/qvip_top/mgc_apb3_0
PCLK
PRESETn
PADDR
PSEL
PENABLE
PWRITE
PWDATA
Console
> change instance /top/mgc_apb3_0
Info: Create instance: New instance '/top/mgc_apb3_0' of VIP type 'amba/apb3' created successfully.
> change test qvip_top
Ready
guest@orw-daerne-v...
configurator [2016062...

5.6.1 Hierarchy of the block_c Environment

The following diagram shows the hierarchy of block_c:



The standard protocol agents generated by the QVIP configurator are contained by the UVMF environment named qvip_agents_environment. This environment is integrated into the block_c_environment using the qvip_subenv label in YAML or the addQvipSubEnv() Python API in the environment code generator. The numbers on the diagram are used to identify the UVM connections and the generator constructs used to create the connections.

5.6.2 Using the Configurator Tool

The configurator uses a GUI to customize environments containing custom and standard protocols. Click the "Protocols" tab to add protocols, which can then be customized on the right side of the configurator. To change the name of the testbench, right click on the whitespace under the "Testbench" tab on the left. The name of the test bench must be changed from the default name of 'top' to avoid recursive module instantiation. In this example we are changing the test bench name from 'top' to 'qvip_agents'. When the desired settings are completed, click the "Generate Models/ Testbench" button. The configurator will create a folder containing the output (In the picture below it will be called "qvip_agents_dir"). Inside this folder there will be a folder

titled “uvmf,” which will contain the UVMF based configurator output. Set the environment variable QVIP_AGENTS_DIR_NAME so that it points to the uvmf folder. Inside of the sv pkg file within the “uvmf” folder are the Python API based addQvipSubEnv and addQvipBfm functions. These functions are provided as comments in the package file. These functions already have the arguments filled out, and are ready to be copy pasted into the environment and bench python config files respectively. The addQvipSubEnv() function and addQvipBfm() function in the block_c_env_config.py and block_c_ben_config.py were generated by the QVIP configurator.

Beginning with the 10.7a release of Questa VIP, the YAML file for the generated environment will be in a separate file in the uvmf diredtory.

5.6.2.1 Connecting to a QVIP analysis_port using addQvipConnection

The YAML qvip_connections and Python API addQvipConnection function connects the QVIP analysis_port to an outside component. Please see the YAML or Python API reference manual for details. The environment instantiation name is prepended to the QVIP agent instantiation name. In the example below the QVIP axi4_master_0 agent within the qvip_env environment has an analysis_port referenced by trans_ap. This analysis_port is connected to the axi4_master_0_ae analysis_export within the blk_c_pred component.

Python API format:

```
env.addQvipConnection('qvip_env_axi4_master_0','trans_ap','blk_c_pred','axi4_master_0_ae')
```

YAML format:

```
qvip_connections :
 -  driver: "qvip_env_axi4_master_0"
    ap_key: "trans_ap"
    receiver: "blk_c_pred.axi4_master_0_ae"
```

The analysis ports within QVIP are located within an associative array of analysis ports named *ap*. This associative array is indexed using a string. In the above example the trans_ap argument is used as a string to index into the associative array of analysis ports within the QVIP agent to select the desired analysis port within the array. All analysis ports within *ap* broadcast transactions that are casted to the base class type named mvc_sequence_item_base. These analysis ports must be connected to analysis exports parameterized to receive transactions of type mvc_sequence_item_base. Transactions received through the analysis export must be casted to the expected transaction type.

The QVIP configurator automatically adds analysis ports to agents. The table below shows the analyis port name, string index, and transaction type broadcasted from the analysis port. Remember that the analysis export must be parameterized to receive

mvc_sequence_item_base and the received transaction must be casted to the type listed in the table below.

Protocol	analysis_port string index	Broadcasted transaction type
AHB	burst_transfer	ahb_master_burst_transfer #(AHB_NUM_MASTERS, AHB_NUM_MASTER_BITS, AHB_NUM_SLAVES, AHB_ADDRESS_WIDTH, AHB_WDATA_WIDTH, AHB_RDATA_WIDTH)
APB3	trans_ap	apb3_host_apb3_transaction #(SLAVE_COUNT, ADDRESS_WIDTH , WDATA_WIDTH , RDATA_WIDTH)
AXI	trans_ap	axi_master_rw_transaction #(AXI_ADDRESS_WIDTH , AXI_RDATA_WIDTH , AXI_WDATA_WIDTH , AXI_ID_WIDTH)
AXI4	trans_ap	axi4_master_rw_transaction #(AXI4_ADDRESS_WIDTH, AXI4_RDATA_WIDTH, AXI4_WDATA_WIDTH, AXI4_ID_WIDTH, AXI4_USER_WIDTH, AXI4_REGION_MAP_SIZE)
AXI4 Streaming	packet_ap	axi4stream_master_packet #(AXI4_ID_WIDTH, AXI4_USER_WIDTH, AXI4_DEST_WIDTH, AXI4_DATA_WIDTH)
	transfer_ap	axi4stream_master_transfer #(AXI4_ID_WIDTH, AXI4_USER_WIDTH, AXI4_DEST_WIDTH, AXI4_DATA_WIDTH)
Can	rx_frame_ap	can_node_rx_frame
Cec	cec_frm_ap	cec_intf_frame
DP	trans_ap_aux_pkt	dp_source_aux_pkt
Ethernet	data_frame_ap	ethernet_device_data_frame

	control_frame_ap	ethernet_device_control_frame
	fault_seq_ap	ethernet_device_fault_sequence
	data_frame_ap_facing	mvc_facing_end_item#(ethernet_device_data_frame)
	control_frame_ap_facing	mvc_facing_end_item#(ethernet_device_control_fra me)
Ethernet 1g	Data_frame_ap	eth_1g_device_data_frame
	Control_frame_ap	eth_1g_device_control_frame
HDMI	trans_ap_frame	hdmi_tx_frame
HSI	Frame_ap	hsi_dev_frame
	rx_frame_ap	hsi_dev_frame
I2C	i2c_xfer_item	i2c_master_i2c_data_transfer
I2S	i2s_xfr_ap	i2s_transmitter_transfer
I3C	i3c_rx_msg_ap	i3c_dev_rx_msg
	i3c_tx_msg_ap	i3c_dev_tx_msg
Interlaken	rx_pkt_ap	interlaken_device_rx_pkt
	tx_pkt_ap	interlaken_device_tx_pkt
JESD204	device_samples_ap	jesd204_tx_device_samples
JTAG	jtag_data_ap	jtag_master_load_data #(SYSTEM_INPUT_PINS, SYSTEM_OUTPUT_PINS, CONFIG_NUM_OF_BSR_REG, CONFIG_IR_WIDTH)
	jtag_instruction_ap	jtag_master_load_instruction #(SYSTEM_INPUT_PINS, SYSTEM_OUTPUT_PINS, CONFIG_NUM_OF_BSR_REG, CONFIG_IR_WIDTH)
MIL1553	msg_bc2rt_ap	mil_1553_bc_rxn_msg
	msg_mode_cmd_ap	mil_1553_bc_mode_cmd_msg
	msg_rt2bc_ap	mil_1553_bc_txn_msg
	msg_rt2rt_ap	mil_1553_bc_rt2rt_msg
PCI Initiator	trans_txn_ap	pci_dev_txn
PCIe See note below regarding *prefix*	*prefix*_us_rx_cmpl_ap	pcie_device_end_facing_completion #(LANES, PIPE_BYTES_MAX, CONFIG_NUM_OF_FUNCTIONS)
	prefix _us_rx_mal_tlp_ap	pcie_device_end_facing_malformed_tlp #(LANES, PIPE_BYTES_MAX, CONFIG_NUM_OF_FUNCTIONS)
	prefix _us_rx_req_ap	pcie_device_end_facing_request #(LANES, PIPE_BYTES_MAX,

		CONFIG_NUM_OF_FUNCTIONS)
	prefix _us_tx_cmpl_ap	pcie_device_end_completion #(LANES, PIPE_BYTES_MAX, CONFIG_NUM_OF_FUNCTIONS)
	prefix _us_tx_mal_tlp_ap	pcie_device_end_facing_completion #(LANES, PIPE_BYTES_MAX, CONFIG_NUM_OF_FUNCTIONS)
	prefix _us_tx_req_ap	pcie_device_end_request #(LANES, PIPE_BYTES_MAX, CONFIG_NUM_OF_FUNCTIONS)
Smartcard	PTS_req_and_rsp_ap	smartcard_if_end_PTS_req_resp
	T_0_data_xfer_ap	smartcard_if_end_T_0_xfer
	T_1_data_xfer_ap	smartcard_if_end_T_1_xfer
	activate_card_ap	smartcard_if_end_activate_contact
	card_answer_to_reset_ap	smartcard_if_end_card_atr
	card_reset_ap	smartcard_if_end_card_reset
	data_in_ap	facing_end_data_t
	data_out_ap	smartcard_if_end_data_char
	deactivate_card_ap	smartcard_if_end_deactivate_contact
Spacewire	rcvd_pkt_ap	mvc_facing_end_item #(spacewire_node_pkt)
	trans_pkt_ap	spacewire_node_pkt
SPI	spi_master_transaction_ap	spi_master_data_transfer #(SPI_SS_WIDTH)
SPI4	burst_ap	mvc_item_listener #(spi4_device_burst_transfer #(SPI4_MAX_PORT, SPI4_CALENDAR_LENGTH_0, SPI4_CALENDAR_LENGTH_1, SPI4_CALENDAR_LENGTH_0_REPROVISION, SPI4_CALENDAR_LENGTH_1_REPROVISION, SPI4_DATA_WIDTH))
	burst_ap_rcvd	mvc_item_listener #(spi4_device_burst_transfer #(

		SPI4_MAX_PORT, SPI4_CALENDAR_LENGTH_0, SPI4_CALENDAR_LENGTH_1, SPI4_CALENDAR_LENGTH_0_REPROVISION, SPI4_CALENDAR_LENGTH_1_REPROVISION, SPI4_DATA_WIDTH))
	calendar_ap	mvc_item_listener #(spi4_device_calendar_transfer #(SPI4_MAX_PORT, SPI4_CALENDAR_LENGTH_0, SPI4_CALENDAR_LENGTH_1, SPI4_CALENDAR_LENGTH_0_REPROVISION, SPI4_CALENDAR_LENGTH_1_REPROVISION, SPI4_DATA_WIDTH))
	calendar_ap_rcvd	mvc_item_listener #(spi4_device_calendar_transfer #(SPI4_MAX_PORT, SPI4_CALENDAR_LENGTH_0, SPI4_CALENDAR_LENGTH_1, SPI4_CALENDAR_LENGTH_0_REPROVISION, SPI4_CALENDAR_LENGTH_1_REPROVISION, SPI4_DATA_WIDTH))
	dp_training_ap	mvc_item_listener #(spi4_device_data_path_training #(SPI4_MAX_PORT, SPI4_CALENDAR_LENGTH_0, SPI4_CALENDAR_LENGTH_1, SPI4_CALENDAR_LENGTH_0_REPROVISION, SPI4_CALENDAR_LENGTH_1_REPROVISION, SPI4_DATA_WIDTH))
	dp_training_ap_rcvd	mvc_item_listener #(spi4_device_data_path_training #(SPI4_MAX_PORT, SPI4_CALENDAR_LENGTH_0, SPI4_CALENDAR_LENGTH_1, SPI4_CALENDAR_LENGTH_0_REPROVISION, SPI4_CALENDAR_LENGTH_1_REPROVISION, SPI4_DATA_WIDTH)

		)
	packet_ap	mvc_item_listener #(spi4_device_packet_transfer #(SPI4_MAX_PORT, SPI4_CALENDAR_LENGTH_0, SPI4_CALENDAR_LENGTH_1, SPI4_CALENDAR_LENGTH_0_REPROVISION, SPI4_CALENDAR_LENGTH_1_REPROVISION, SPI4_DATA_WIDTH))
	packet_ap_rcvd	mvc_item_listener #(spi4_device_packet_transfer #(SPI4_MAX_PORT, SPI4_CALENDAR_LENGTH_0, SPI4_CALENDAR_LENGTH_1, SPI4_CALENDAR_LENGTH_0_REPROVISION, SPI4_CALENDAR_LENGTH_1_REPROVISION, SPI4_DATA_WIDTH))
	status_training_ap	mvc_item_listener #(spi4_device_status_training #(SPI4_MAX_PORT, SPI4_CALENDAR_LENGTH_0, SPI4_CALENDAR_LENGTH_1, SPI4_CALENDAR_LENGTH_0_REPROVISION, SPI4_CALENDAR_LENGTH_1_REPROVISION, SPI4_DATA_WIDTH))
	status_training_ap_rcvd	mvc_item_listener #(spi4_device_status_training #(SPI4_MAX_PORT, SPI4_CALENDAR_LENGTH_0, SPI4_CALENDAR_LENGTH_1, SPI4_CALENDAR_LENGTH_0_REPROVISION, SPI4_CALENDAR_LENGTH_1_REPROVISION, SPI4_DATA_WIDTH))
UART	data_in	mvc_facing_end_item #(uart_device_end_data_char)
	data_out	uart_device_end_data_char

The *prefix* value for the pcie analysis port names is the PCIe agent name preceeded by pcie_. For example: a PCIe agent named uP_rc would have the *prefix* value of pcie_uP_rc.

5.6.3 Clock generation within QVIP Configurator generated UVMF module

The QVIP configurator generates a module that contains all of the QVIP interface BFM's. The name of this module is based on the test bench name: hdl_*benchName*.sv. This module contains a default clock and reset generator. There is no interaction between this clock/reset generator and the clock/reset generator used in hdl_top.sv created by UVMF. If a single clock/reset generator is required then ports can be added to the hdl_*benchName* module to bring in clock and reset from hdl_top.sv.

5.6.4 Block to top reuse of QVIP Configurator generated environment

Block to top reuse of QVIP Configurator generated code is supported beginning with the 10.7a release of QVIP. The Configurator generated UVMF based environment is reusable as a sub-environment at any environment level with any number of instantiations.

5.6.5 Block to top reuse of QVIP Configurator generated module

Block to top reuse of QVIP Configurator generated code is supported beginning with the 10.7a release of QVIP. The Configurator generated UVMF based module contains a single parameter named UNIQUE_ID, a parameter for each BFM that sets the ACTIVE/PASSIVE setting for each BFM, and a parameter named EXT_CLK_RESET that determines if the Configurator generated clock and reset driver is used.

The UNIQUE_ID string parameter is prepended to the BFM interface name. The result is used as the field_name argument to the uvm_config_db::set call to place each interface BFM in the uvm_config_db.

The parameter for each BFM that sets the ACTIVE/PASSIVE setting is named <AGENT_NAME>_ACTIVE where <AGENT_NAME> is the capitalized name of the QVIP agent. When set to 1, this parameter sets the BFM in ACTIVE mode. When set to 0, this parameter sets the BFM in PASSIVE mode.

The EXT_CLK_RESET parameter determines if the default clock and reset generators within the Configurator generated module are instantiated. When set to 1, then the clock and reset generators are not instantiated. This is considered EXTernal clock and reset generation. The clock and reset signals from hdl_top.sv must be connected to the Configurator generated modules clock and reset signals. When set to 0, then the clock and reset generators are instantiated within the Configurator generated module and connected to all QVIP BFM's within the module.

6 Resource Sharing within the UVM Framework

6.1 Overview

The UVM Framework uses uvm_config_db as the mechanism for sharing resources within the test. The resources shared include virtual interface handles, sequencer handles and agent configuration handles. The information used to place resources into and retrieve resources from the uvm_config_db are listed in the resource table. The resource table is described in the following section.

It is important to note that the code listed below is automatically generated when using the python scripts to generate an interface package, environment package or project bench. The table below can be created from the python generated code for use by test writers.

The example table and code below is from the ahb2spi example.

6.2 The Resource Table

The resource table is used to collect all information required to access resources associated with an interface. The "Port Description" column is a listing and description of each interface in the design that is either actively driven or passively monitored. The "Port Type" column identifies the name of the BFM interfaces for the port. They are the declared name of the interface BFMs. The "Transaction Type" column identifies the transaction type that is sent to the sequencer for that interface. The "Port Name" column lists a unique string variable for each interface listed. The "Port Name" variable must be unique from all other "Port Name" values in the column.

Port Description	Port Type	Transaction Type	Port Name
AHB Interface	ahb_monitor_bfm ahb_driver_bfm	ahb_transaction	"AHB_BFM"
WB Interface	wb_monitor_bfm	wb_transaction	"AHB2WB_WB_BFM"
WB Interface	wb_monitor_bfm	wb_transaction	"WB2SPI_WB_BFM"
SPI Interface	spi_monitor_bfm spi_driver_bfm	spi_transaction	"SPI_BFM"

6.3 Sharing and Accessing Environment Resources

6.3.1 Virtual Interface Handles

The resource table has the information needed to access virtual interface handles for the monitor BFMs and driver BFMs. The fields used are in red and green. The "Port Type" field is used in the uvm_config_db Type field. The scope for all virtual interface handles is 'null, UVMF_VIRTUAL_INTERFACES'. This creates a generic scope for all virtual interface handles. The "Port Name" field is used for the uvm_config_db Field name variable. This provides a unique identifier to differentiate between multiple BFM entries of the same type.

Interface Resource Sharing – AHB2SPI Example

Port Description	Port Type	Transaction Type	Port Name
AHB Interface	ahb_monitor_bfm ahb_driver_bfm	ahb_transaction	"AHB_BFM"
WB Interface	wb_monitor_bfm	wb_transaction	"AHB2WB_WB_BFM"
WB Interface	wb_monitor_bfm	wb_transaction	"AHB2WB_WB_BFM"
SPI Interface	spi_monitor_bfm spi_driver_bfm	spi_transaction	"SPI_BFM"

- Top module places virtual interface into uvm_config_db
  - Type: ahb_monitor_bfm or ahb_driver_bfm
  - Scope: null , UVMF_VIRTUAL_INTERFACES
  - Field name: "AHB_BFM"
- To retrieve virtual interface from uvm_config_db
  - Type: ahb_monitor_bfm or ahb_driver_bfm
  - Scope: null , UVMF_VIRTUAL_INTERFACES
  - Field name: "AHB_BFM"

51

The code for placing a virtual interface handle into the uvm_config_db is shown below. It is located in hdl_top.sv of the ahb2wb example project_bench.

```
 94    initial begin // tbx vif_binding_block
 95     import uvm_pkg::uvm_config_db;
 96
 97     uvm_config_db #( virtual ahb_monitor_bfm )::
 98        set( null , UVMF_VIRTUAL_INTERFACES , AHB_BFM , ahb_mon_bfm );
 99     uvm_config_db #( virtual ahb_driver_bfm  )::
100        set( null , UVMF_VIRTUAL_INTERFACES , AHB_BFM , ahb_drv_bfm  );
101
102     uvm_config_db #( virtual wb_monitor_bfm )::
103        set( null , UVMF_VIRTUAL_INTERFACES  , AHB2WB_WB_BFM  , ahb2wb_wb_mon_bfm );
104     uvm_config_db #( virtual wb_monitor_bfm )::
105        set( null , UVMF_VIRTUAL_INTERFACES  , WB2SPI_WB_BFM  , wb2spi_wb_mon_bfm );
106
107     uvm_config_db #( virtual spi_monitor_bfm )::
108        set( null , UVMF_VIRTUAL_INTERFACES , SPI_BFM , spi_mon_bfm );
109     uvm_config_db #( virtual spi_driver_bfm  )::
110        set( null , UVMF_VIRTUAL_INTERFACES , SPI_BFM , spi_drv_bfm  );
111
112    end
```

The retrieval of the virtual interface handle is located in the agent configuration. The code below is from uvmf_parameterized_agent_configuration_base.svh

```
106  virtual function void initialize(
107                                       uvmf_active_passive_t activity,
108                                       string agent_path,
109                                       string interface_name);
110    active_passive      = activity;
111    this.interface_name = interface_name;
112
113    // Checking the config_db
114    void'(uvm_config_db #(uvm_bitstream_t)::
115      get(null,interface_name,"enable_transaction_viewing",enable_transaction_viewing));
116
117   if( !uvm_config_db #( MONITOR_BFM_BIND_T )::
118     get( null , UVMF_VIRTUAL_INTERFACES , interface_name , monitor_bfm ) )
119           `uvm_error("Config Error" ,
120            $sformatf("uvm_config_db #( MONITOR_BFM_BIND_T )::get cannot find resource %s",interface_name) )
121
122   if ( activity == ACTIVE ) begin
123      if( !uvm_config_db #( DRIVER_BFM_BIND_T )::
124       get( null , UVMF_VIRTUAL_INTERFACES , interface_name , driver_bfm ) )
125           `uvm_error("Config Error" ,
126            $sformatf("uvm_config_db #( DRIVER_BFM_BIND_T )::get cannot find resource %s",interface_name) )
127   end
128
129  endfunction
```

6.3.2 Sequencer Handles

The resource table has the information needed to access sequencer handles in the simulation. The fields used are in red and green. The “Transaction Type” field is used in the uvm_config_db Type field. The scope for all virtual interface handles is ‘null, UVMF_SEQUENCERS’. This creates a generic scope for all sequencer handles. The “Port Name” field is used for the uvm_config_db Field name variable. This provides a unique identifier to differentiate between multiple sequencer entries of the same type.

Sequencer Resource Sharing – AHB2SPI Example

Port Description	Port Type	Transaction Type	Port Name
AHB Interface	ahb_monitor_bfm ahb_driver_bfm	ahb_transaction	"AHB_BFM"
WB Interface	wb_monitor_bfm	wb_transaction	"AHB2WB_WB_BFM"
WB Interface	wb_monitor_bfm	wb_transaction	"AHB2WB_WB_BFM"
SPI Interface	spi_monitor_bfm spi_driver_bfm	spi_transaction	"SPI_BFM"

- Agent places interface sequencer into uvm_config_db
  - Type: uvm_sequencer#(ahb_transaction)
  - Scope: null , UVMF_SEQUENCERS
  - Field name: "AHB_BFM"
- To retrieve sequencer from uvm_config_db
  - Type: uvm_sequencer#(ahb_transaction)
  - Scope: null , UVMF_SEQUENCERS
  - Field name: "AHB_BFM"

52

The code for placing a sequencer handle into the uvm_config_db is shown below. It is located in uvmf_parameterized_agent.svh.

```
136       uvm_config_db #( sequencer_t )::
137          set( null , UVMF_SEQUENCERS , configuration.interface_name, sequencer );
138       driver = DRIVER_T::type_id::create({agent_name,"_driver"},this);
```

The retrieval of the sequencer handle is located in the top level sequence. The code below is from ahb2spi_sequence_base.svh

```
60    if( !uvm_config_db #( uvm_sequencer #(ahb_transaction) )::
61      get( null , UVMF_SEQUENCERS , AHB_BFM , ahb_sequencer ) )
62            `uvm_error("Config Error" ,
63             "uvm_config_db #( uvm_sequencer#(ahb_transaction) )::get cannot find resource ahb_sequ
64    if( !uvm_config_db #( uvm_sequencer #(spi_transaction) )::
65      get( null , UVMF_SEQUENCERS , SPI_BFM , spi_sequencer ) )
66            `uvm_error("Config Error" ,
67             "uvm_config_db #( uvm_sequencer#(spi_transaction) )::get cannot find resource spi_sequ
```

6.3.3 Agent Configuration Handles

The agent configuration can be retrieved from the uvm_config_db using the same value from the "Port Name" column of the resource table. The scope for all agent configurations is 'null, UVMF_CONFIGURATIONS'. The code below is from

ahb_configuration.svh. The configuration places itself in the uvm_config_db using the interface name variable in line 82.

```
74  // ****************************************************************************
75      virtual function void initialize(uvmf_active_passive_t activity,
76                                       string agent_path,
77                                       string interface_name);
78
79       super.initialize( activity, agent_path, interface_name);
80
81       uvm_config_db #( ahb_configuration )::set( null ,agent_path,       UVMF_AGENT_CONFIG, this );
82       uvm_config_db #( ahb_configuration )::set( null ,UVMF_CONFIGURATIONS, interface_name, this );
83
84      endfunction
```

The code below is from ahb2spi_sequence_base.svh. The ahb agents configuration class is retrieved from uvm_config_db in lines 69-70.

```
69      if( !uvm_config_db #( ahb_configuration )::
70        get( null ,UVMF_CONFIGURATIONS, AHB_BFM, ahb_config ) )
71              `uvm_error("Config Error" ,
72                "uvm_config_db #( ahb_configuration )::get cannot find resource ahb_config" )
```

6.4 Turning on transaction viewing for selected interfaces

The resource table can be used to enable transaction viewing for selected interfaces without recompilation of batch mode simulations. The “Port Name” field of the resource table is used to enable transaction viewing for a single, group or all interfaces. Use the UVM command line processing line shown below as an example. The plusarg is added to the vsim command.

Enabling Interface Transaction Viewing

Port Description	Port Type	Transaction Type	Port Name
AHB Interface	ahb_monitor_bfm ahb_driver_bfm	ahb_transaction	"AHB_BFM"
WB Interface	wb_monitor_bfm	wb_transaction	"AHB2WB_WB_BFM"
WB Interface	wb_monitor_bfm	wb_transaction	"AHB2WB_WB_BFM"
SPI Interface	spi_monitor_bfm spi_driver_bfm	spi_transaction	"SPI_BFM"

- Use the UVM Command Line Interface to turn on transaction viewing for:
  - A selected interface:
    +uvm_set_config_int="AHB_BFM",enable_transaction_viewing,1
  - Interface groups:
    +uvm_set_config_int="AHB2*",enable_transaction_viewing,1
  - All interfaces:
    +uvm_set_config_int=*,enable_transaction_viewing,1

53

7 Environment and Interface Initialization within the UVM Framework

7.1 Overview

The UVM Framework is architected for reuse. One key characteristic of reuse is self-containment. Reusable components automatically get needed resources, construct and configure children components and make internal resources available to other components. The two mechanisms for applying this are the initialize function and set_config function.

The initialize function passes information down through the configuration object hierarchy. This information configures environments and agents in a simulation. It begins at the top level UVM test and ends at agent configurations. It is the mechanism by which all agents are initialized.

The set_config function passes configuration object handles down through the environment hierarchy.

It is important to note that the code listed below is automatically generated when generating an interface package, environment package and project bench.

7.2 Top-down initialization through the initialize function

The initialize function passes information down through the configuration hierarchy. It starts at the top level UVM test and ends at the agent configurations. The configuration class for each agent in a simulation is initialized using this mechanism. At the top level UVM test and environment level the initialize function passes the following information: simulation level, hierarchical path down to the configurations environment and an array of string names that uniquely identify each interface in the design. At the agent level the initialize function passes the following information: active/passive state of the agent, hierarchical path to the configurations agent and unique string identifying the agents handle to the interface BFMs (driver BFM and monitor BFM).

The following initialize flow is from the ahb2wb example.

The code below is from test_top.svh. Keep in mind that the build_phase function of this component completes before the environment build_function is executed. This allows for the configurations to be constructed and initialized before the environment hierarchy is constructed.

This function performs the following flow.

Lines 30-33: Create an array of strings that contains the unique names of each interface BFMs in the simulation. These values are parameters defined in tb/parameters

Lines 35-38: Create an array of agent activity settings. The order matches the interface name order. Since this is a block level simulation, all agents are passive. The bench generator creates this code and assumes all agents are active. The user must specify which agents are passive.

Line 68: Pass the simulation level to the environment, uvm_test_top.environment. This argument can be used to set agent activity levels. However, a much more flexible mechanism is to use the array of agent activity levels provided in line 72. The UVMF generators use the array of agent activity settings.

Line 69: Pass the path to the environment, uvm_test_top.environment. Sub-environments will append child component hierarchical names to this path and pass the result down through the initialize function.

Line 70: Pass the array of agent interface names to the environment, uvm_test_top.environment. Each environment will distribute these names to sub-environemnts. Ultimately, each agent configuration will receive its own unique interface name. The agent configuration uses this name to retrieve interface BFM handles from the uvm_config_db. This name is also used by the agent to place its sequencer in the uvm_config_db for retrieval by the top level sequence.

Line 71: Pass the register model handle to the environment, uvm_test_top.environment. This handle is the register block for this level of environment. The ahb2wb environment does not contain a register model so the value is null.

Line 72: Pass the agent activity settings to the environment, uvm_test_top.environment. . Each environment will distribute these settings to sub-environemnts. Ultimately, each agent configuration will receive its own unique setting. The agent uses this value in its configuration to determine whether or not to construct its sequencer and driver. A passive agent does not have a driver BFM.

```
30  string interface_names[] = {
31      ahb_pkg_ahb_BFM /* ahb      [0] */ ,
32      wb_pkg_wb_BFM /* wb      [1] */
33  };
34
35  uvmf_active_passive_t interface_activities[] = {
36      ACTIVE /* ahb      [0] */ ,
37      ACTIVE /* wb      [1] */
38  };
39
```

```
64   virtual function void build_phase(uvm_phase phase);
65
66     super.build_phase(phase);
67     configuration.initialize(
68                              BLOCK,
69                              "uvm_test_top.environment",
70                              interface_names,
71                              null,
72                              interface_activities);
73   endfunction
```

The code below is from ahb2wb_env_configuration.svh. It is executed by test_top.svh. It is passed the information required by the environemt to configure sub-environments and agents.

This function performs the following:

Line 89: Super.initialize is called to perform tasks required by all environment configurations. It also contains debug code that is executed when the simulation verbosity is set to UVM_DEBUG.

Liine 97: The initialize function is called to pass the ahb agent configuration information required for setup. This call includes the active/passive setting, full path to the agent in the environment hierarchy, and the unique string name of this interface.

Line 98: The activity setting determines the ACTIVE or PASSIVE setting of this agent.

Line 99: The full path to the agent in the environment hierarchy that this agent configuration is paired with. Each agent has its own corresponding agent configuration. This path is used by the agent configuration to place itself in the uvm_config_db. The path is used to set the scope of the uvm_config_db::set call so that only one agent receives this agent configuration.

Line 100: The interface name uniquely identifying this agent. This variable is used by the agent configuration to retrieve the monitor BFM and driver BFM if the agent is configured as ACTIVE.

Lines 103-107: These lines are used to pass initialization information to the wb agent.

```
81    function void initialize(
82                              uvmf_sim_level_t sim_level,
83                              string environment_path,
84                              string interface_names[],
85                              uvm_reg_block register_model = null,
86                              uvmf_active_passive_t interface_activity[] = null
87                              );
88
89      super.initialize(
90                              sim_level,
91                              environment_path,
92                              interface_names,
93                              register_model,
94                              interface_activity
95                              );
96
97      ahb_config.initialize(
98                              interface_activity[0],
99                              {environment_path,".ahb"},
100                             interface_names[0]
101                             );
102
103     wb_config.initialize(
104                             interface_activity[1],
105                             {environment_path,".wb"},
106                             interface_names[1]
107                             );
108
109   endfunction
```

The code below is from ahb_configuration.svh. It is executed by ahb2wb_configuration.svh. It is passed the active/passive state, the hierarchy down to the agent, "uvm_test_top.environment.a_agent", and the interface name.

This function performs the following flow:

Line 79: Call to super.initialize to execute the initialize function in uvmf_parameterized_agent_configuration_base.svh. The flow of this function is described in the next code section.

Line 81: This configuration object places itself in the uvm_config_db for the agent identified by agent_path to retrieve.

Line 82: This configuration object places itself in the uvm_config_db using the string that identifies the interface BFMs used for this agent. This creates an automatic association between an interface and its configuration.

```
74  // ****************************************************************************
75    virtual function void initialize(uvmf_active_passive_t activity,
76                                     string agent_path,
77                                     string interface_name);
78
79     super.initialize( activity, agent_path, interface_name);
80
81     uvm_config_db #( ahb_configuration )::set( null ,agent_path,       UVMF_AGENT_CONFIG, this );
82     uvm_config_db #( ahb_configuration )::set( null ,UVMF_CONFIGURATIONS, interface_name, this );
83
84    endfunction
```

The code below is from uvmf_parameterized_agent_configuration_base.svh. It is executed by ahb_configuration.svh. It is passed the active/passive state, the hierarchy down to the agent, "uvm_test_top.environment.a_agent", and the interface name.

This function performs the following flow:

Lines 110-111: Set the local activity level and interface string name variables from the arguments to the function call.

Lines 114-115: Check the uvm_config_db for command line setting of the enable_transaction_viewing flag for this interface.

Lines 117-120: Retrieve the handle to the monitor BFM. Generate an error if the get function fails.

Lines 122-126: If the agent is configured as active then retrieve the handle to the driver BFM. Generate an error if the get function fails.

```
106 virtual function void initialize(
107                                     uvmf_active_passive_t activity,
108                                     string agent_path,
109                                     string interface_name);
110    active_passive      = activity;
111    this.interface_name = interface_name;
112
113    // Checking the config_db
114    void'(uvm_config_db #(uvm_bitstream_t)::
115      get(null,interface_name,"enable_transaction_viewing",enable_transaction_viewing));
116
117  if( !uvm_config_db #( MONITOR_BFM_BIND_T )::
118    get( null , UVMF_VIRTUAL_INTERFACES , interface_name , monitor_bfm ) )
119          `uvm_error("Config Error" ,
120            $sformatf("uvm_config_db #( MONITOR_BFM_BIND_T )::get cannot find resource %s",interface_name) )
121
122  if ( activity == ACTIVE ) begin
123     if( !uvm_config_db #( DRIVER_BFM_BIND_T )::
124      get( null , UVMF_VIRTUAL_INTERFACES , interface_name , driver_bfm ) )
125          `uvm_error("Config Error" ,
126            $sformatf("uvm_config_db #( DRIVER_BFM_BIND_T )::get cannot find resource %s",interface_name) )
127  end
128
129 endfunction
```

7.3 Debug features for identifying interface_name array issues

Debug tracing of the interface_name array is provided using messaging located in the UVMF base code. This feature is enabled when setting the test verbosity to UVM_DEBUG. This can be done when using the UVMF Makefiles by adding TEST_VERBOSITY=UVM_DEBUG to the make command line. The following trace information is displayed:

For each environment:

- Hierarchical path
- The name and activity for each interface passed to the environment

For each agent in the environment:

- Hierarchical path
- Interface name and activity

7.4 Top-down passing of environment configuration through the set_config function.

All environments should be extended from the uvmf_environment_base. The set_config function from this class is shown below. This function is used by test_top and all lower environments to pass in the environments configuration handle.

```
49    // FUNCTION : set_config
50    function void set_config( CONFIG_T cfg );
51        configuration = cfg;
52    endfunction
```

The following set_config flow is from the ahb2spi example.

The code below is from uvmf_test_base.svh. This call is made in the build_phase and happens automatically for any test_top derived from uvmf_test_base. This call is performed on the top level environment. If the environment is not a block level, i.e. chip level or above, then the set_config for lower level environments must be done by the top level environment as shown in the next code example.

```
93      environment.set_config(configuration);
```

The code below is from ahb2spi_environment.svh. The build phase of this chip level environment uses set_config to pass configuration handles into lower level environments.

This function performs the following flow:

Line 59: Construct the ahb2wb environment.

Line 60: Pass the ahb2wb_config handle into the ahb2wb environment using set_config.

Line 62: Construct the wb2spi environment.

Line 63: Pass the wb2spi_config handle into the wb2spi environment using set_config.

```
55  // ****************************************************************************
56    virtual function void build_phase(uvm_phase phase);
57      super.build_phase(phase);
58
59      ahb2wb_env = ahb2wb_environment::type_id::create("ahb2wb_env",this);
60      ahb2wb_env.set_config( configuration.ahb2wb_config );
61
62      wb2spi_env = wb2spi_environment::type_id::create("wb2spi_env",this);
63      wb2spi_env.set_config( configuration.wb2spi_config );
64
65    endfunction
```

8 Enabling Transaction Viewing within the UVM Framework

8.1 Overview

The code that is responsible for transaction viewing in the waveform viewer is in three locations: agent configuration, agent monitor and transaction class. The agent configuration contains a variable that turns transaction viewing on and off. The agent monitor creates the transaction viewing stream and calls the function in the transaction class that adds the transaction to the stream. The transaction class adds itself to the transaction stream.

8.2 UVM Framework transaction viewing flow

8.2.1 Creating a transaction stream

The transaction stream is a handle to which all transactions to be viewed are added. This stream is created in the monitor extended from the uvmf_monitor_base. The following code in uvmf_monitor_base creates the stream.

```
// FUNCTION: start_of_simulation_phase
virtual function void start_of_simulation_phase(uvm_phase phase);
   if (configuration.enable_transaction_viewing)
     transaction_viewing_stream = $create_transaction_stream({"..",get_full_name(),".","txn_stream"});
 endfunction
```

The stream is automatically created in the start_of_simulation_phase and is conditional on the enable_transaction_viewing flag in the agent configuration. The name of the stream is the full hierarchical path to the monitor with ".txn_stream" appended. This stream can be found in the Questa object window when viewing objects within the monitor.

8.2.2 Adding a transaction to the stream

Transactions to be viewed in the waveform viewer must be added to a transaction stream. The function that adds a transaction is located in the transaction class. The following code from the run_phase of the uvmf_monitor_base calls the add_to_wave

function of the transaction class. The add_to_wave function adds the trans class to the transaction stream.

```
if ( configuration.enable_transaction_viewing )
 trans.add_to_wave(transaction_viewing_stream);
```

The add_to_wave function of the wb_transaction class of the wb_pkg is shown below.

```
56 //------------------------------------------------------------------------
57    virtual function void add_to_wave(int transaction_viewing_stream_h);
58     if ( transaction_view_h == 0)
59        transaction_view_h = $begin_transaction(transaction_viewing_stream_h,op.name(),start_time);
60       if      ( op == WB_RESET ) $add_color( transaction_view_h, "red" );
61       else if ( op == WB_WRITE ) $add_color( transaction_view_h, "blue" );
62       else if ( op == WB_READ )  $add_color( transaction_view_h, "green" );
63       super.add_to_wave(transaction_view_h);
64       $add_attribute(transaction_view_h, op, "op");
65       $add_attribute(transaction_view_h, addr, "addr");
66       $add_attribute(transaction_view_h, data, "data");
67       $add_attribute(transaction_view_h, byte_select, "byte_select");
68       $end_transaction(transaction_view_h,end_time);
69       $free_transaction(transaction_view_h);
70    endfunction
```

The following is a description of the operations performed by the add_to_wave function:

Lines 58-59: The transaction_view_h is a handle to the transaction viewing object for this transaction within the transaction stream. Each transaction in the stream has a unique transaction viewing handle. If the handle is null then a new handle is generated using the begin_transaction system call. The start_time argument of the begin_transaction call determines the start time of the transaction in the waveform viewer.

Lines 60-62: These lines set the color of the transaction within the waveform viewer based on the transaction operation type.

Line 63: This line executes the add_to_wave function in the base class. It adds variables in the base class, uvmf_transaction_base, to the waveform viewer.

Lines 64-67: The add_attribute system function adds transaction variables to the waveform viewer. The second argument is the data variable to be added. The third argument identifies the variable value.

Line 68: The end_transaction system function call sets the end time of the transaction in the waveform viewer.

Line 69: The free_transaction system function call closes the transaction viewing handle on the stream and completes the process of adding the transaction.

8.3 Switches for enabling transaction viewing

8.3.1 UVM Reporting Detail setting

The UVM recording detail of the simulation can be set in either of the two mechanisms listed below:

Add the following line to the UVM test case in any phase prior to and including the run_phase:
set_config_int("*","recording_detail",UVM_FULL);

Add the following line to the vsim command line:
+uvm_set_config_int=*,recording_detail,UVM_FULL

8.3.2 Command line switches

-uvmcontrol=all –msgmode both
+uvm_set_config_int=*,enable_transaction_viewing,1

8.3.3 Adding transaction viewing stream to the waveform viewer

The line below adds the transaction viewing stream, txn_stream, in the wishbone agent monitor component with the hierarchical path listed. The transaction stream of any UVMF based monitor can be added to the waveform in the same manner.

add wave –noupdate /uvm_root/uvm_test_top/environment/wb_agent/wb_agent_monitor/txn_stream

9 The Top Level Modules

9.1.1 Hdl_top

The module named hdl_top, located in tb/test bench directory, contains the signal bundle interfaces, interface BFMs and DUT. It also includes the uvm_config_db::set calls to pass virtual interface handles to the UVM. The UVMF uses a two top architecture, hdl_top and hvl_top, to support emulation. Hdl_top is synthesized into the emulator. Hvl_top is run on the host simulator.

9.1.1.1 Instantiating Interfaces

A UVMF interface is divided into three pieces; the signal bundle, monitor BFM and driver BFM. Each instance of a protocol interface requires an instantiation of the signal bundle and monitor BFM. Each active instance of a protocol interface requires the instantiation of a driver BFM. The following code is from the instantiation of the interfaces in hdl_top from the ALU example.

```
48  alu_out_if          alu_out_bus(.clk(),.rst(),.done(),.result());
49  alu_out_monitor_bfm alu_out_mon_bfm(alu_out_bus);
50  alu_out_driver_bfm  alu_out_drv_bfm(alu_out_bus);
51
52  alu_in_if           alu_in_bus(.clk(),.rst(),.valid(),.ready(),.op(),.a(),.b());
53  alu_in_monitor_bfm  alu_in_mon_bfm(alu_in_bus);
54  alu_in_driver_bfm   alu_in_drv_bfm(alu_in_bus);
```

Lines 48 and 52 instantiate the alu_in_if and alu_out_if signal bundles respectively. Lines 49 and 53 instantiate the alu_in_if and alu_out_if monitor BFMs. Lines 50 and 54 instantiate the alu_in_if and alu_out_if driver BFMs. The monitor and driver BFM port lists require a reference to the signal bundle. This allows the monitor BFM to observe signals in the signal bundle and the driver BFM to drive signals in the signal bundle.

9.1.1.2 Instantiating a Verilog DUT

A verilog DUT is instantiated using the following format. The code is from the instantiation of the DUT in hdl_top from the ALU example.

```
59  alu   #(.OP_WIDTH(8), .RESULT_WIDTH(16)) DUT  (
60        .clk    (alu_in_bus.clk ) ,
61        .rst    (alu_in_bus.rst ) ,
62        .ready  (alu_in_bus.ready ) ,
63        .valid  (alu_in_bus.valid ) ,
64        .op     (alu_in_bus.op ) ,
65        .a      (alu_in_bus.a ) ,
66        .b      (alu_in_bus.b ) ,
67        .done   (alu_out_bus.done ) ,
68        .result (alu_out_bus.result ) );
```

Line 59 defines the module type, alu, and instance name, DUT. Parameters for the module are defined in the parenthesis following the #. Lines 60 through 68 list the ports of the alu module. The parenthesis following each port name identify the signal to be connected to the port. For each port connection the interface signal bundle and signal within the signal bundle is identified using hierarchical notation.

9.1.1.3 Instantiating a VHDL DUT

The format for instantiating a VHDL DUT is identical to instantiating a verilog DUT with the exception of line 59. In line 59 the work library, VHDL entity and VHDL architecture must be specified. The following line replaces line 59:

\workLib.entity(architecture) #(.OP_WIDTH(8), .RESULT_WIDTH(16)) DUT (

9.1.2 Hvl_top

The module named hvl_top, located in tb/test bench directory, imports the test package and contains the call to run_test which executes the UVM phases. The UVMF uses a two top architecture, hdl_top and hvl_top, to support emulation. Hdl_top is synthesized into the emulator. Hvl_top is run on the host simulator.

The code below is from hvl_top from the ALU example.

```
41  import uvm_pkg::*;
42  import alu_test_pkg::*;
43
44  module hvl_top;
45
46    initial run_test();
47
48  endmodule
```

Line 42 imports the alu test package which contains all alu tests. The call to run_test in line 46 starts execution of all UVM phases.

10 Creating Test Scenarios

10.1 Overview

A test scenario is a series of stimulus used to configure and stimulate the design. A test scenario can be created by writing a new test, a new sequence or both. If the desired stimulus does not exist in a sequence package then a new sequence must be created. The new sequence can be selected using either a new test or using the UVM command line processor. This section describes the creation of a new sequence, creation of a new test and how to select the sequence using either the new test or the UVM command line processor.

10.2 Creating a New Sequence

10.2.1 Creating a New Interface Sequence

If a bus operation needs to be created that is protocol specific and not design specific then a new interface sequence should be created. The new sequence should be extended from the _sequence_base class located in the interface package. The new sequence should be added to the interface package. The steps below describe how to create a new interface sequence and add the sequence to the package. It assumes the name of the interface package is abc_pkg and that the name of the new sequence is new_sequence.

1) In the abc_pkg/src folder create a file named new_sequence.svh
2) In new_sequence.svh add a class that extends abc_sequence_base. At a minimum this sequence should contain a factory registration macro and constructor.
3) Add the desired behavior to this sequence.
4) Include the new sequence in abc_pkg.sv after the inclusion of abc_sequence_base.svh

10.2.2 Creating a New Environment Sequence

If a sequence needs to be created that is design specific and may be reused at block and chip level simulation then a new environment sequence should be created. The new sequence should be extended from the _sequence_base class located in the environment package. The new sequence should be added to the environment package. The steps below describe how to create a new environment sequence and add the sequence to the package. It assumes the name of the environment package is abc_env_pkg and that the name of the new sequence is new_sequence.

1) In the abc_env_pkg/src folder create a file named new_sequence.svh
2) In new_sequence.svh add a class that extends abc_env_sequence_base. At a minimum this sequence should contain a factory registration macro and constructor.
3) Add the desired behavior to this sequence.

4) Include the new sequence in abc_env_pkg.sv after the inclusion of abc_env_sequence_base.svh

10.2.3 Creating a New Top Level Sequence

The top level sequence controls the flow and order of all lower level sequences. If a sequence needs to be created that is design specific and will not be reused at block and chip level simulation then a new top level sequence should be created. The new sequence should be extended from the _sequence_base class located in the sequence package of the bench. The new sequence should be added to the sequence package. The steps below describe how to create a new top level sequence and add the sequence to the package. It assumes the name of the bench sequence package is abc2def_sequences_pkg and that the name of the new sequence is new_sequence.

1) In the tb/sequences/abc2def_sequences/src folder create a file named new_sequence.svh
2) In new_sequence.svh add a class that extends abc2def_sequence_base. At a minimum this sequence should contain a factory registration macro and constructor.
3) Add the desired behavior to this sequence.
4) Include the new sequence in abc2def_sequences_pkg.sv after the inclusion of abc2def_sequence_base.svh

An example sequence derived from the top level sequence base is generated by the bench generator. It is named example_derived_test_sequence and is located in project_benches/<bench_name>/tb/sequences/src.

10.3 Creating a New Test

All UVMF test packages have a test named test_top. This test contains the top level configuration, top level environment and top level sequence. Test_top constructs and connects the components. It also starts the sequence identified in the class definition of test_top. A new test case is created by extending test_top. The steps below describe how to create a new test class. It assumes the name of the test package is abc2def_test_pkg and the name of the new test is new_test. Each example in UVMF has an example derived test named example_derived_test.

1) In the tb/tests/src folder create a new file named new_test.svh
2) In new_test.svh add a class that extends test_top. At a minimum this test should contain a factory registration macro, constructor and build_phase function.
3) Add the desired factory overrides to this sequence in the build_phase. The factory overrides should be prior to super.build_phase(phase).
4) Include the new test in abc2def_test_pkg.sv after the inclusion of test_top.svh

10.4 Selecting a New Test Scenario using the UVM Factory

10.4.1 Using a New Test Class

The TEST_NAME makefile variable is used to select the test. This variable is used to set the UVM_TESTNAME command line variable. The default value for the TEST_NAME variable is test_top. To select another test add TEST_NAME=new_test to the make command line.

10.4.2 Using the UVM Command Line Processor

The UVM command line processor can be used to override any class or object within the simulation. This includes overriding sequences. To select a new test scenario by performing an override using the UVM command line processor add the following to the vsim command line:
+uvm_set_type_override=requested_type_class_name,override_type_class_name

11 Adding non UVMF components to an Existing UVMF Bench and Environment

11.1 Adding a non-UVMF based agent

The recommended method of adding a non-UVMF based agent to a UVMF environment is to wrap the agent within a UVMF generated environment. The envioronment only contains the non-UVMF based agent and the glue code to configure and initialize the agent when the initialize function of the environment is called.

The generated UVMF Makefile will need to be updated to compile the encapsulated agent.

Encapsulating the non-UVMF agent within a UVMF environment enables the use of the non-UVMF agent with the UVMF environemnt generator.

11.2 Adding a non-UVMF based environment

The recommended method of adding a non-UVMF based environment to a UVMF environment is to wrap the non-UVMF environment within a UVMF generated environment. The envioronment only contains the non-UVMF based environment and the glue code to configure and initialize the agent when the initialize function of the environment is called.

The generated UVMF Makefile will need to be updated to compile the encapsulated environment.

Encapsulating the non-UVMF environment within a UVMF environment enables the use of the non-UVMF environment with the UVMF environment generator.

11.3 Adding a QVIP agent

The recommended method of adding QVIP agents to a UVMF environment be performed using the QVIP Configurator. The QVIP Configurator is a graphical tool for selecting and configuring standard protocols. The QVIP Configurator generates a UVMF environment that contains all selected agents.

12 Making a non-UVMF Interface VIP Compatible with UVMF

Non-UVMF VIP can be made UVMF compatible with some modifications. The requirements listed below are necessary in order to be compatible with UVMF's UVM reuse methodology. They are also required for seamless use with UVMF environment and test bench generators. The requirements below are written assuming a non UVMF interface VIP protocol named xyz.

12.1 Interface Package

All classes, typedefs, and class specializations used in the interface VIP must be contained within a package with a _pkg suffix. For example, xyz_pkg. This package must contain classes such as driver, monitor, coverage, agent, transaction, sequences, etc.

This package must contain the following imports:

```
import uvmf_base_pkg_hdl::*;
import uvmf_base_pkg::*;
```

12.2 Transaction Class

The transaction class must have an _transaction suffix. For example, xyz_transaction. This can be done using a typedef. For example, typedef my_xyz_transaction_name xyz_transaction;

The transaction class must be extended from uvmf_transaction_base. A class that extends uvm_sequence_item can be changed to extend from uvmf_transaction_base because uvmf_transaction_base extends uvm_sequence_item.

12.3 Configuration Class

The configuration class must have an _configuration suffix. For example, xyz_configuration. This can be done using a typedef. For example, typedef my_xyz_ocnfiguration_name xyz_configuration;

The configuration class must have a convert2string implementation. This is because UVMV environment configurations convert2string will call convert2string of all agent configurations within the enviornment configuration.

The configuration class must have an initialization function with the following prototype:

```
virtual function void initialize(    uvmf_active_passive_t activity,
                                     string agent_path,
                                     string interface_name);
```

The activity argument will have a value of ACTIVE or PASSIVE based on the agent activity. If the agent is actively driving stimulus into the DUT then the agent is ACTIVE. If the agent is only passively monitoring bus traffic then the agent is PASSIVE. The agent_path argument is the full path to this configurations agent. Since the UVMF architecture dictates that the configuration classes are not within the environment hierarchy the agent_path must be used to pass the configuration handle to its agent. The interface_name argument is a unique string identifying the resources associated with this agent. The resources include its virtual interface handles(s), configuration class handle, and sequencer class handle.

12.4 Agent Class

The agent class must have an _agent_t suffix. For example, xyz_agent_t. This can be done using a typedef. For example, typedef my_xyz_agent_name xyz_agent_t;

The agent class must have an analysis_port named monitored_ap which broadcasts transactions of type xyz_transaction as described in the "Transaction Class" section.

If the agent is ACTIVE then it must place its sequencer handle in the uvm_config_db using the interface_name argument of the configurations initialize call as the field_name argument of the uvm_config_db::set call with the following scope: null, UVMF_SEQUENCERS.

12.5 Interface

The interfaces should include a driver BFM, monitor BFM and signal bundle. The names of these should have _driver_bfm, _monitor_bfm, and _if suffixes respectively. If this recommendation is not followed then the interface instantiations within the generated hdl_top.sv must be modified accordingly.

12.6 Makefile

A makefile named Makefile should be created for compiling the interface and package for the VIP. The makefile should contain a make target named comp_xyz_pkg where xyz is the protocol name. If this recommendation is not followed then the test bench makefile must be modified accordingly. The makefile from an interface package generated using the UVMF code generator can be used as a template for creating the interface makefile.

12.7 Directory Structure

The interface VIP should have the following directory structure given the protocol name of xyz:

xyz_pkg which contains the xyz_pkg declaration and Makefile
xyz_pkg/src which contains all other source files

The xyz_pkg should be located in the following location: $(UVMF_VIP_LIBRARY_HOME)/interface_packages/.

If this recommendation is not followed then the test bench makefile must be modified accordingly.

13 Appendix A: UVM classes used within UVMF

13.1 Overview

In order to minimize risk and minimize the UVM learning curve, a minimum subset of classes from the uvm_pkg have been used. Most of the classes used in UVMF were contained in the predecessor of the UVM, ovm_pkg.

13.2 UVM Component Classes Used

uvm_test, uvm_env, uvm_agent, uvm_sequencer, uvm_driver, uvm_monitor, uvm_subscriber, uvm_scoreboard, uvm_analysis_port, uvm_port_component_base, uvm_port_list, uvm_report_server, uvm_reg_predictor

13.3 UVM Data Classes Used

uvm_object, uvm_sequence_item, uvm_sequence, uvm_tlm_analysis_fifo

13.4 UVM Phases Used

Build_phase, Connect_phase, End_of_elaboration_phase, Start_of_simulation_phase, Run_phase, Extract_phase, Check_phase, Report_phase

13.5 UVM Macros Used

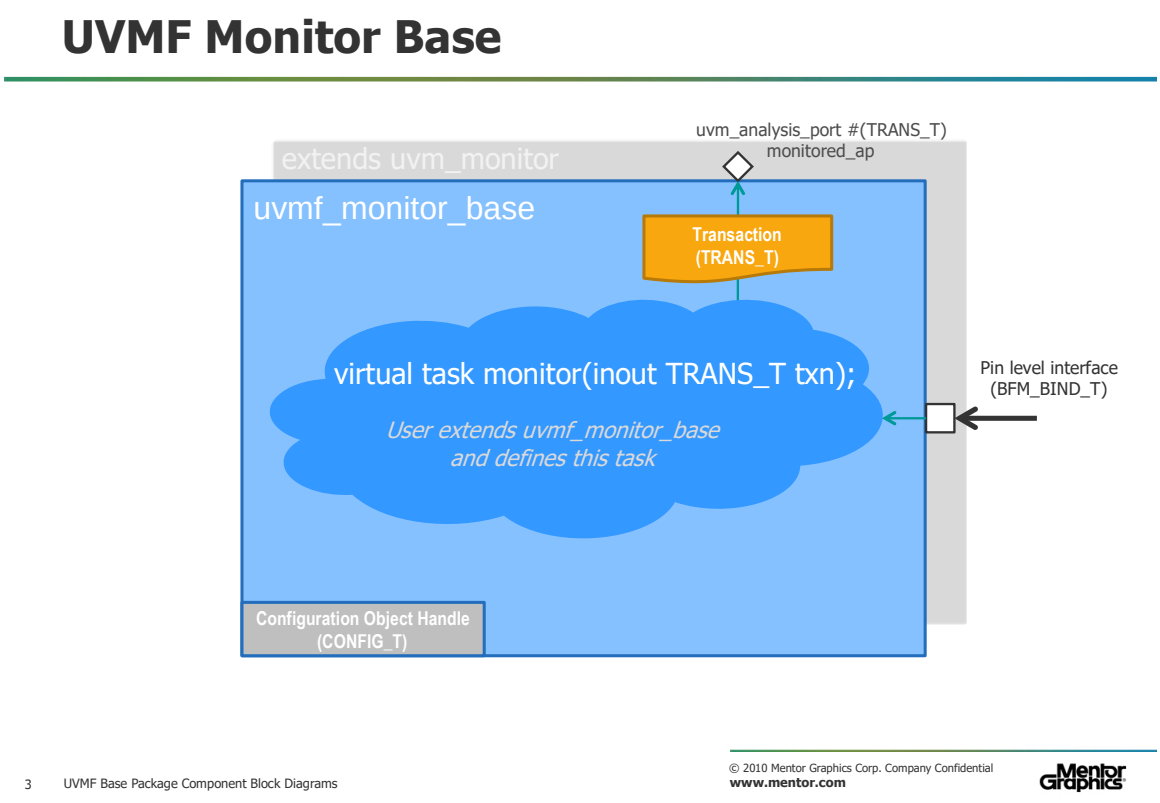
uvm_component_utils, uvm_component_param_utils, uvm_object_utils, uvm_object_param_utils, uvm_info, uvm_warning, uvm_error, uvm_fatal, uvm_analysis_imp_decl

13.6 Miscellaneous UVM Features Used

uvm_config_db, UVM factory, Phase_ready_to_end, Raise_objection, Drop_objection

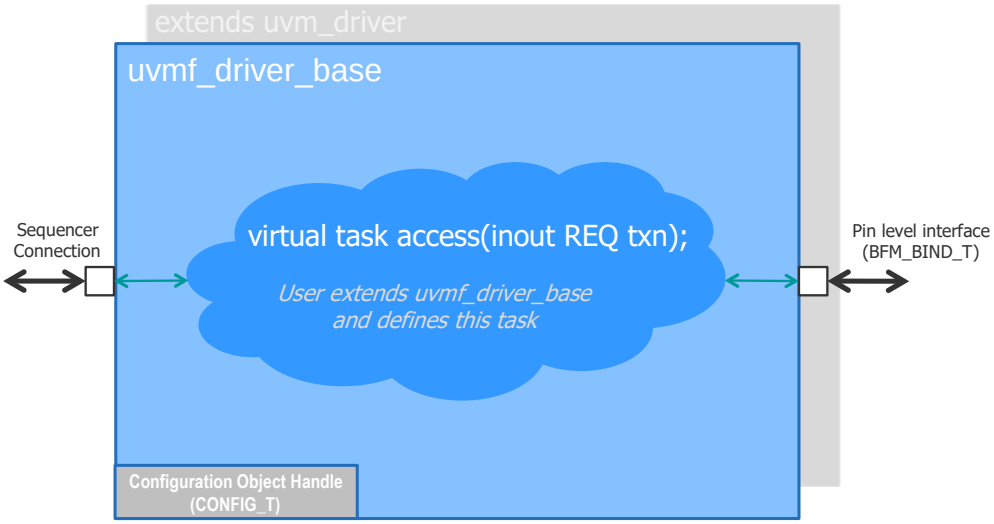
14 Appendix B: UVMF Base Package Block Diagrams

14.1 Monitor Base



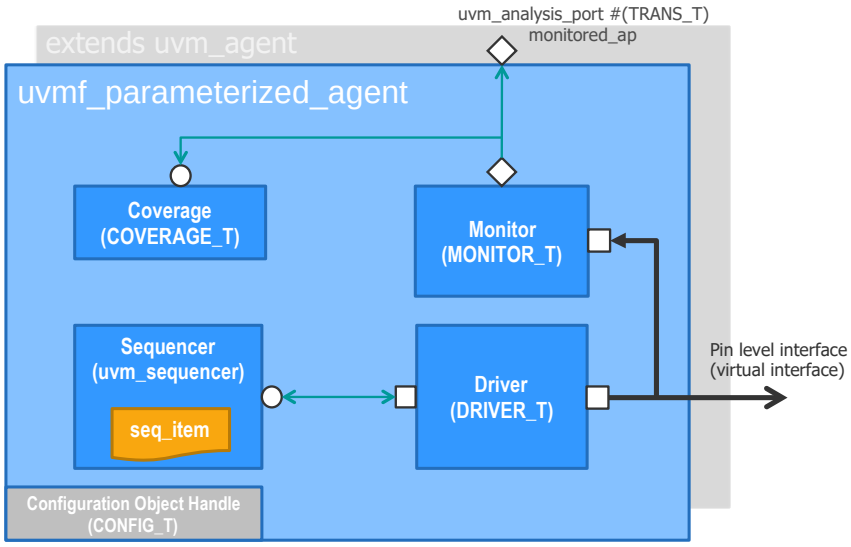
14.2 Driver Base

UVMF Driver Base



14.3 Parameterized Agent

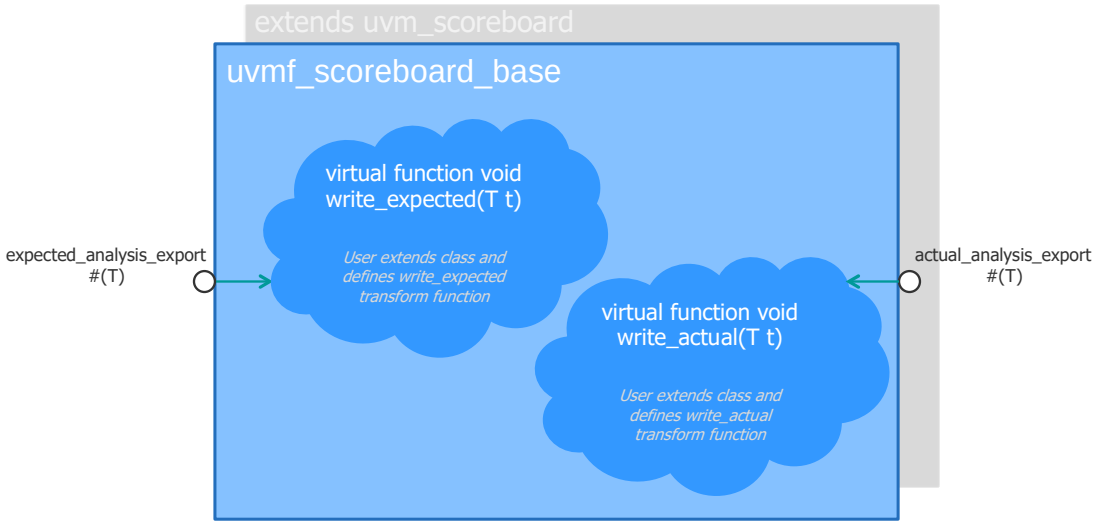
UVMF Parameterized Agent



2 UVMF Base Package Component Block Diagrams

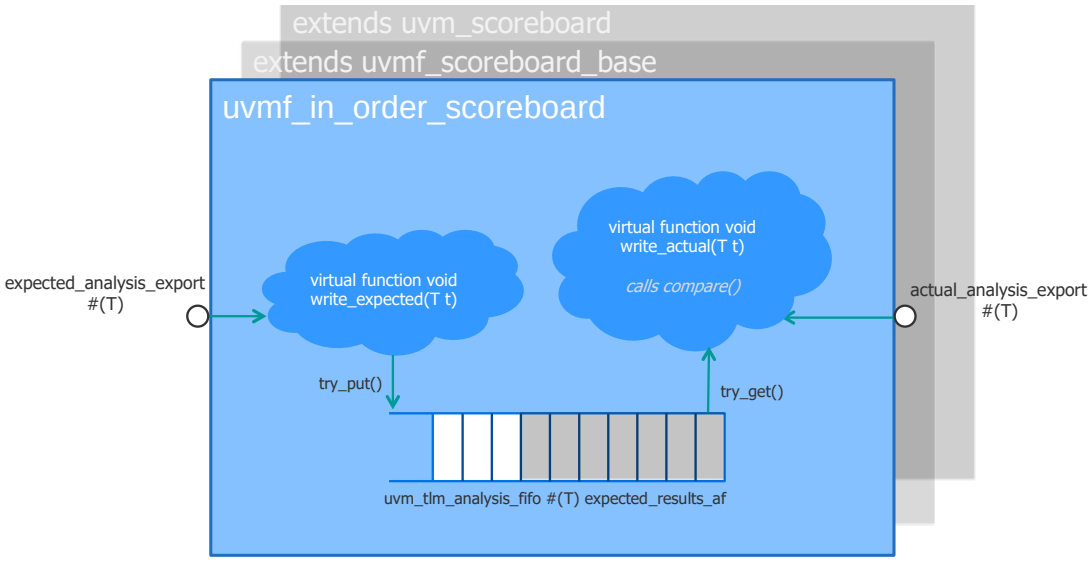
14.4 Scoreboard Base

UVMF Scoreboard Base



14.5 In Order Scoreboard

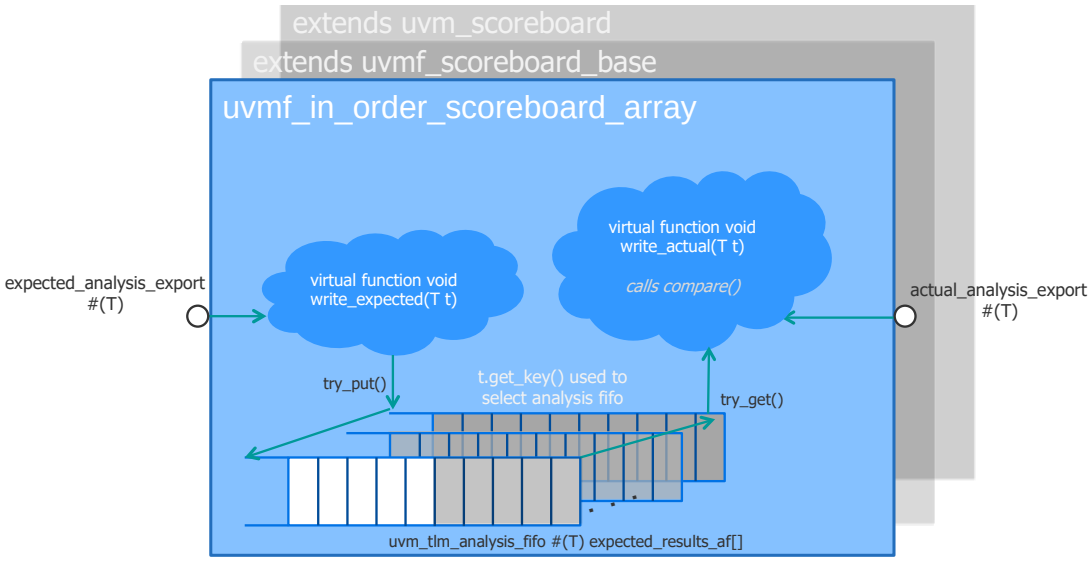
UVMF In-Order Scoreboard



12 UVMF Base Package Component Block Diagrams

14.6 In Order Scoreboard Array

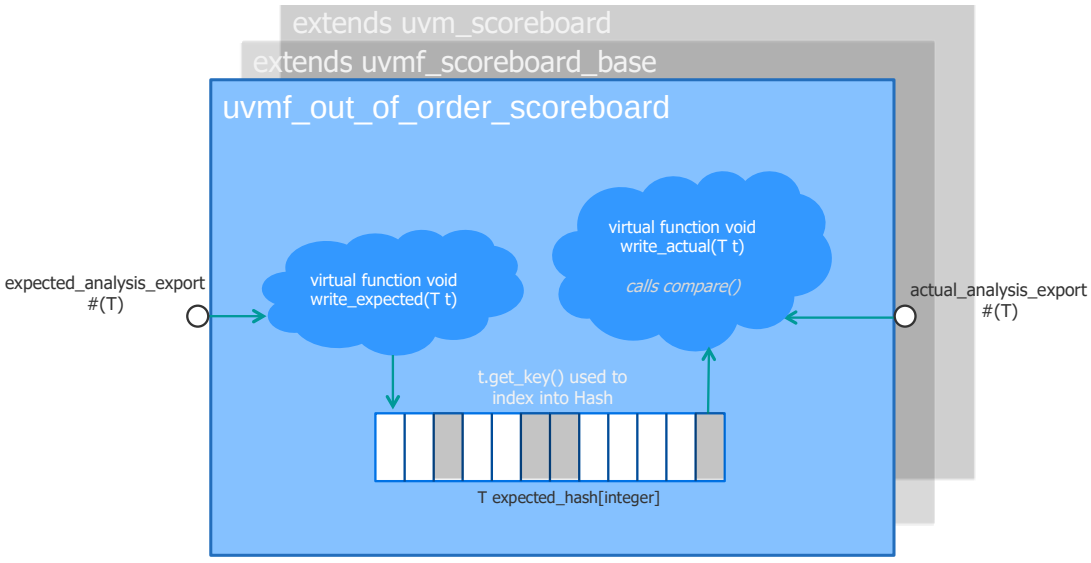
UVMF In-Order Scoreboard Array



13 UVMF Base Package Component Block Diagrams

14.7 Out of Order Scoreboard

UVMF Out-of-Order Scoreboard



14 UVMF Base Package Component Block Diagrams

14.8 In Order Race Scoreboard